MCUXpresso SDK Documentation

Release 25.12.00

NXP
Dec 18, 2025

Table of contents

1 KW45B41Z-EVK **3**
1.1 Overview . . . 3
1.2 Getting Started with MCUXpresso SDK Package . . . 3
1.2.1 Getting Started with Package . . . 3
1.3 Getting Started with MCUXpresso SDK GitHub . . . 20
1.3.1 Getting Started with MCUXpresso SDK Repository . . . 20
1.4 Release Notes . . . 44
1.4.1 MCUXpresso SDK Release Notes . . . 44
1.5 ChangeLog . . . 54
1.5.1 MCUXpresso SDK Changelog . . . 54
1.6 Driver API Reference Manual . . . 146
1.7 Middleware Documentation . . . 146
1.7.1 Wireless Bluetooth LE host stack and applications . . . 146
1.7.2 Wireless zigbee stack . . . 146
1.7.3 Wireless Connectivity Framework . . . 146
1.7.4 Multicore . . . 146
1.7.5 FreeMASTER . . . 147
1.7.6 FreeRTOS . . . 147

2 KW45B41Z83 **149**
2.1 CACHE: LPCAC CACHE Memory Controller . . . 149
2.2 CCM32K: 32kHz Clock Control Module . . . 150
2.3 Clock Driver . . . 160
2.4 CMC: Core Mode Controller Driver . . . 179
2.5 CRC: Cyclic Redundancy Check Driver . . . 191
2.6 EDMA: Enhanced Direct Memory Access (eDMA) Controller Driver . . . 194
2.7 ELEMU: Edgelock Messaging unit driver . . . 213
2.8 EWM: External Watchdog Monitor Driver . . . 217
2.9 FGPIO Driver . . . 220
2.10 C40ESP3 Flash Driver . . . 220
2.11 FlexCAN: Flex Controller Area Network Driver . . . 224
2.12 FlexCAN Driver . . . 224
2.13 FlexCAN eDMA Driver . . . 272
2.14 FlexIO: FlexIO Driver . . . 275
2.15 FlexIO Driver . . . 275
2.16 FlexIO eDMA SPI Driver . . . 292
2.17 FlexIO eDMA UART Driver . . . 296
2.18 FlexIO I2C Master Driver . . . 298
2.19 FlexIO I2S Driver . . . 307
2.20 FlexIO SPI Driver . . . 317
2.21 FlexIO UART Driver . . . 330
2.22 GPIO: General-Purpose Input/Output Driver . . . 341
2.23 GPIO Driver . . . 344
2.24 I3C: I3C Driver . . . 349
2.25 I3C Common Driver . . . 351
2.26 I3C Master Driver . . . 354

2.27 I3C Slave Driver . . . 380
2.28 IMU: Inter CPU Messaging Unit . . . 393
2.29 Common Driver . . . 399
2.30 LIN: Local Interconnect Network Driver . . . 412
2.31 LIN Driver . . . 412
2.32 LIN LPUART Driver . . . 427
2.33 LPADC: 12-bit SAR Analog-to-Digital Converter Driver . . . 435
2.34 LPCMP: Low Power Analog Comparator Driver . . . 456
2.35 LPI2C: Low Power Inter-Integrated Circuit Driver . . . 462
2.36 LPI2C Master Driver . . . 463
2.37 LPI2C Master DMA Driver . . . 477
2.38 LPI2C Slave Driver . . . 479
2.39 LPIT: Low-Power Interrupt Timer . . . 490
2.40 LPSPI: Low Power Serial Peripheral Interface . . . 496
2.41 LPSPI Peripheral driver . . . 496
2.42 LPSPI eDMA Driver . . . 518
2.43 LPTMR: Low-Power Timer . . . 525
2.44 LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver . . . 530
2.45 LPUART Driver . . . 530
2.46 LPUART eDMA Driver . . . 549
2.47 LTC: LP Trusted Cryptography . . . 552
2.48 LTC AES driver . . . 553
2.49 LTC DES driver . . . 558
2.50 LTC HASH driver . . . 567
2.51 LTC PKHA driver . . . 568
2.52 LTC Blocking APIs . . . 577
2.53 MCM: Miscellaneous Control Module . . . 577
2.54 MSCM: Miscellaneous System Control . . . 582
2.55 PORT: Port Control and Interrupts . . . 583
2.56 RTC: Real Time Clock . . . 590
2.57 SEMA42: Hardware Semaphores Driver . . . 597
2.58 SFA: Signal Frequency Analyser . . . 601
2.59 SMSCM: Secure Miscellaneous System Control Module . . . 610
2.60 SPC: System Power Control driver . . . 614
2.61 SYSPM: System Performance Monitor . . . 649
2.62 TPM: Timer PWM Module . . . 652
2.63 TRDC: Trusted Resource Domain Controller . . . 668
2.64 TRGMUX: Trigger Mux Driver . . . 689
2.65 TSTMR: Timestamp Timer Driver . . . 690
2.66 VBAT: Smart Power Switch . . . 691
2.67 VREF: Voltage Reference Driver . . . 698
2.68 WDOG32: 32-bit Watchdog Timer . . . 701
2.69 WUU: Wakeup Unit driver . . . 707

3 Middleware **713**
3.1 Motor Control . . . 713
3.1.1 FreeMASTER . . . 713
3.2 MultiCore . . . 750
3.2.1 Multicore SDK . . . 750
3.3 Wireless . . . 848
3.3.1 NXP Wireless Framework and Stacks . . . 848

4 RTOS **2207**
4.1 FreeRTOS . . . 2207
4.1.1 FreeRTOS kernel . . . 2207
4.1.2 FreeRTOS drivers . . . 2207
4.1.3 backoffalgorithm . . . 2207
4.1.4 corehttp . . . 2207

4.1.5 corejson 2207
4.1.6 coremqtt 2208
4.1.7 corepkcs11 2208
4.1.8 freertos-plus-tcp 2208

This documentation contains information specific to the kw45b41zevk board.

Chapter 1

KW45B41Z-EVK

1.1 Overview

The KW45B41Z EVK is an evaluation kit for KW45 MCUs with 2.4 GHz Bluetooth Low Energy and generic FSK wireless connectivity and CAN/LIN connectivity. The KW45's highly sensitive, optimized 2.4 GHz radio features a PCB F-antenna which can be bypassed to test via SMA connection. The board includes an MCU-Link debug probe, CAN and LIN transceivers, buttons, switches, LEDs and integrated sensors, Arduino shield connectors, a MikroE Click connector and other headers.

MCU device and part on board is shown below:

- Device: KW45B41Z83
- PartNumber: KW45B41Z83AFTA

1.2 Getting Started with MCUXpresso SDK Package

1.2.1 Getting Started with Package

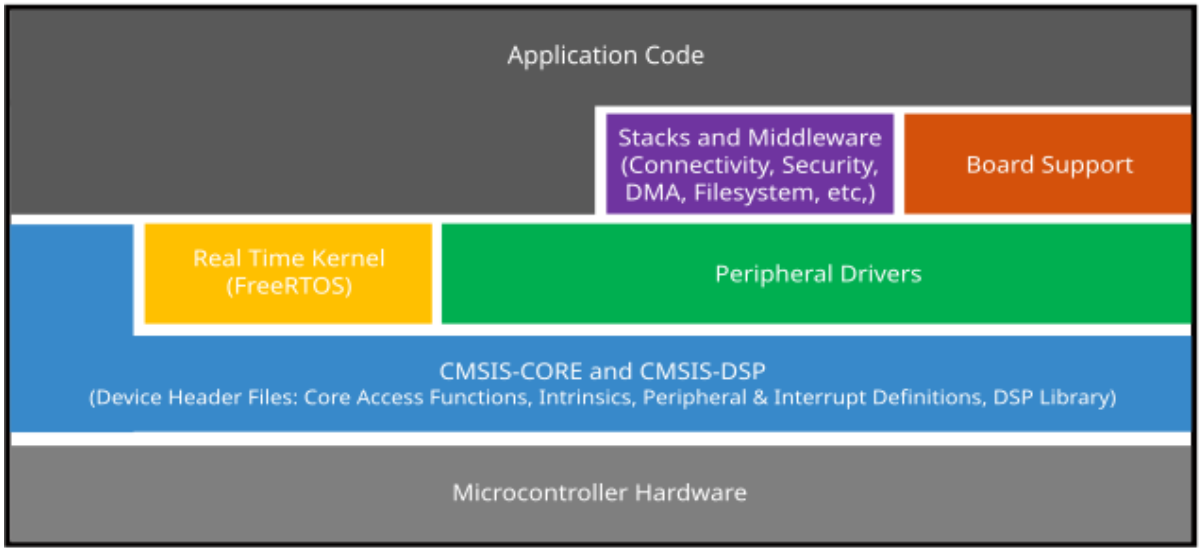
Overview

The NXP MCUXpresso software and tools offer comprehensive development solutions designed to optimize, ease and help accelerate embedded system development of applications based on general purpose, crossover and Bluetooth-enabled MCUs from NXP. The MCUXpresso SDK includes a flexible set of peripheral drivers designed to speed up and simplify development of embedded applications which can be used standalone or collaboratively with the A cores running another Operating System (such as Linux OS Kernel). Along with the peripheral drivers,

the MCUXpresso SDK provides an extensive and rich set of example applications covering everything from basic peripheral use case examples to demo applications. The MCUXpresso SDK also contains optional RTOS integrations such as FreeRTOS and Azure RTOS, device stack, and various other middleware to support rapid development.

For supported toolchain versions, see *MCUXpresso SDK Release Notes for KW45B41Z-EVK* (Document MCUXSDKKW45RN).

For the latest version of this and other MCUXpresso SDK documents, see MCUXpresso Software Development Kit (SDK).



MCUXpresso SDK Board Support Package Folders

MCUXpresso SDK board support package provides example applications for NXP development and evaluation boards for Arm Cortex-M cores including Freedom, Tower System, and LPCXpresso boards. Board support packages are found inside the top-level boards folder and each supported board has its own folder (an MCUXpresso SDK package can support multiple boards). Within each *<board_name>* folder, there are various sub-folders to classify the type of examples it contains. These include (but are not limited to):

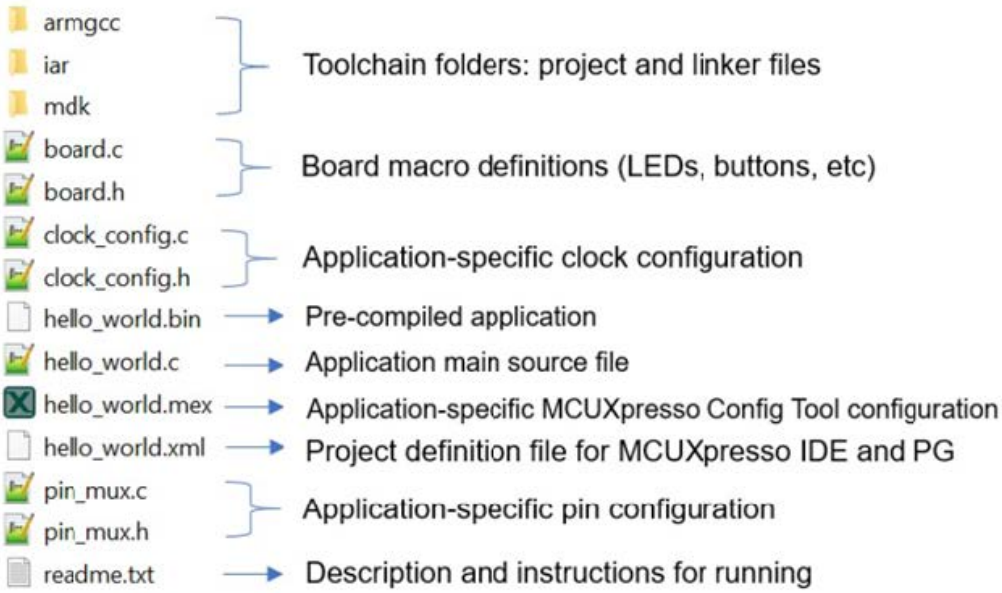
- demo_apps: Full-featured applications that highlight key functionality and use cases of the target MCU. These applications typically use multiple MCU peripherals and may leverage stacks and middleware.
- driver_examples: Simple applications that show how to use the MCUXpresso SDK's peripheral drivers for a single use case. These applications typically only use a single peripheral but there are cases where multiple peripherals are used (for example, SPI conversion using DMA).
- rtos_examples: Basic FreeRTOSTM OS examples that show the use of various RTOS objects (semaphores, queues, and so on) and interfaces with the MCUXpresso SDK's RTOS drivers.
- wireless_examples: Applications that use the Wireless stacks.

Example application structure This section describes how the various types of example applications interact with the other components in the MCUXpresso SDK. To get a comprehensive understanding of all MCUXpresso SDK components and folder structure, see *MCUXpresso SDK API Reference Manual* (document MCUXSDKAPIRM).

Each <board_name> folder in the boards directory contains a comprehensive set of examples that are relevant to that specific piece of hardware. Although we use the hello_world exam-

ple (part of the demo_apps folder), the same general rules apply to any type of example in the <board_name> folder.

In the hello_world application folder, you see the following contents:



All files in the application folder are specific to that example, so it is easy to copy and paste an existing example to start developing a custom application based on a project provided in the MCUXpresso SDK.

Note: To prevent compilation errors, do not use special characters in the path of the SDK such as {!,@,#,$,&,%,^} and space.

Parent topic:*MCUXpresso SDK Board Support Package Folders*

Locating example application source files When opening an example application in any of the supported IDEs, a variety of source files are referenced. The MCUXpresso SDK devices folder is the central component to all example applications. It means the examples reference the same source files and, if one of these files is modified, it could potentially impact the behavior of other examples.

The main areas of the MCUXpresso SDK tree used in all example applications are:

- *devices/<device_name>*: The device's CMSIS header file, MCUXpresso SDK feature file and a few other files
- *devices/<device_name>/drivers*: All of the peripheral drivers for your specific MCU
- *devices/<device_name>/<tool_name>*: Toolchain-specific startup code, including vector table definitions
- *devices/<device_name>/utilities*: Items such as the debug console that are used by many of the example applications
- *devices/<device_name>/project_template*: Project template used by MCUXpresso IDE to create new projects

For examples containing an RTOS, there are references to the appropriate source code. RTOSes are in the rtos folder. The core files of each of these are shared, so modifying one could have potential impacts on other projects that depend on that file.

Parent topic:*MCUXpresso SDK Board Support Package Folders*

Running a Demo Application using IAR

This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK. The hello_world demo application targeted for the **kw45b41zevk** hardware platform is used as an example, although these steps can be applied to any example application in the MCUXpresso SDK.

Build an example application Do the following steps to build the hello_world example application..

1. Open the desired demo application workspace. Most example application workspace files can be located using the following path:

   ```
   <install_dir>/boards/<board_name>/<example_type>/<application_name>/iar
   ```

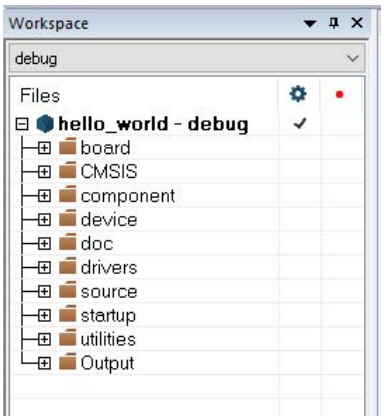
   Using the kw45b41zevk hardware platform as an example, the hello_world workspace is located in:

   ```
   <install_dir>/boards/kw45b41zevk/demo_apps/hello_world/iar/hello_world.eww
   ```

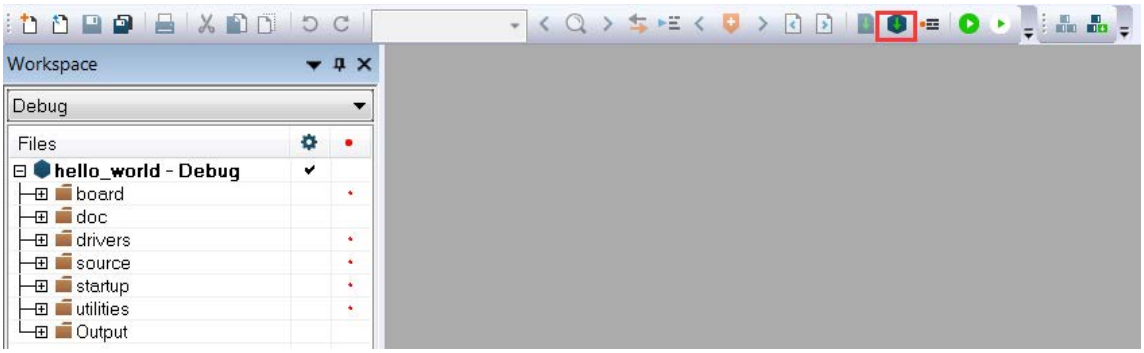
   Other example applications may have additional folders in their path.

2. Select the desired build target from the drop-down menu.

   For this example, select **hello_world – debug**.



3. To build the demo application, click **Make**, highlighted in red in Figure 2.

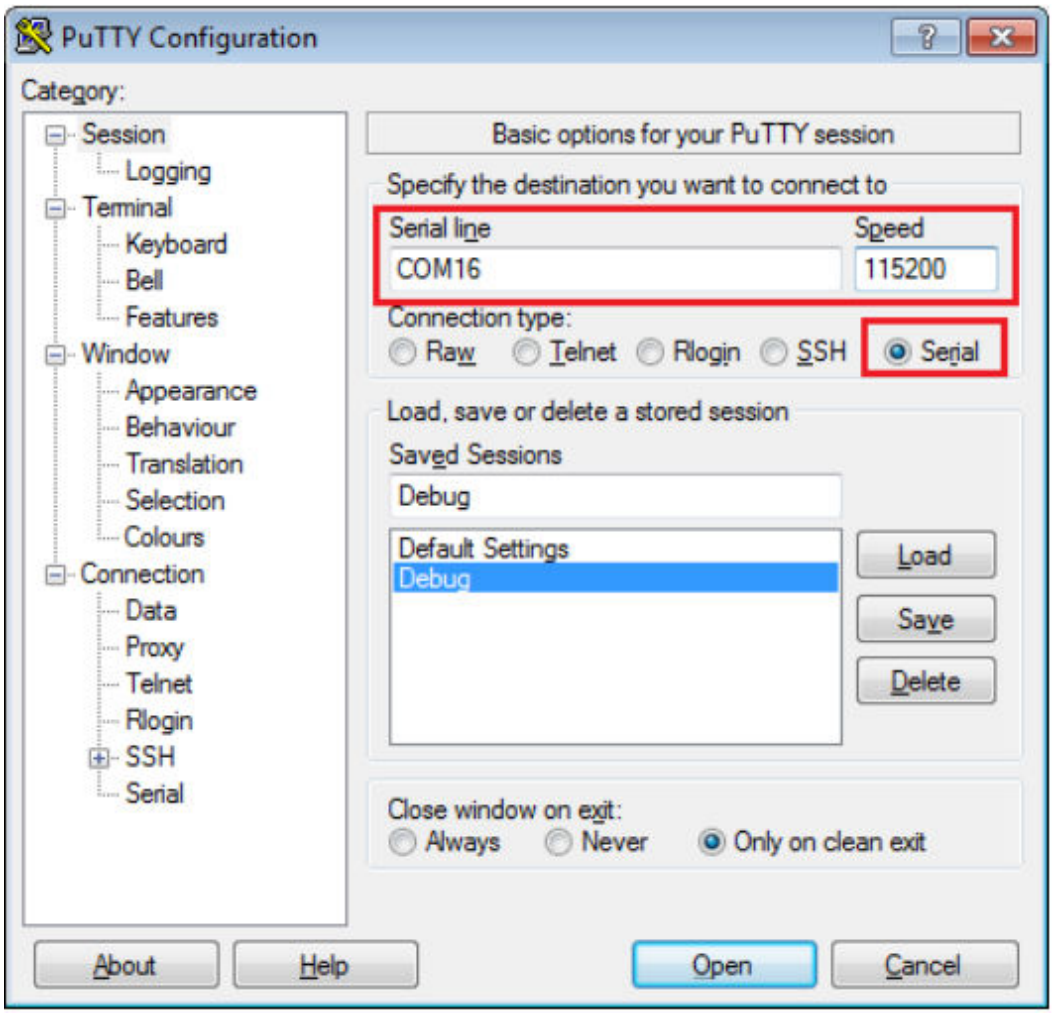


4. The build completes without errors.

Parent topic:*Running a Demo Application using IAR*

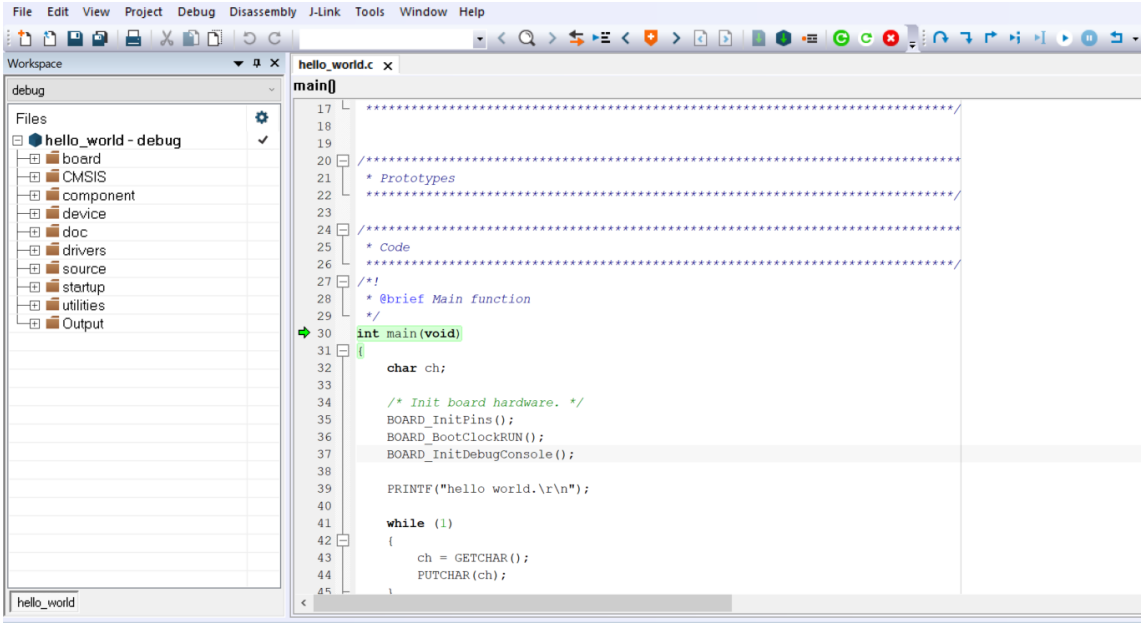
Run an example application To download and run the application, perform these steps:

1. Connect the development platform to your PC via USB cable between the USB connector (J14) and the PC USB connector.
2. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug COM port. To determine the COM port number, see *How to determine COM Port*. Configure the terminal with these settings:
   1. 115200 baud rate, depending on your settings (reference the BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)
   2. No parity
   3. 8 data bits
   4. 1 stop bit



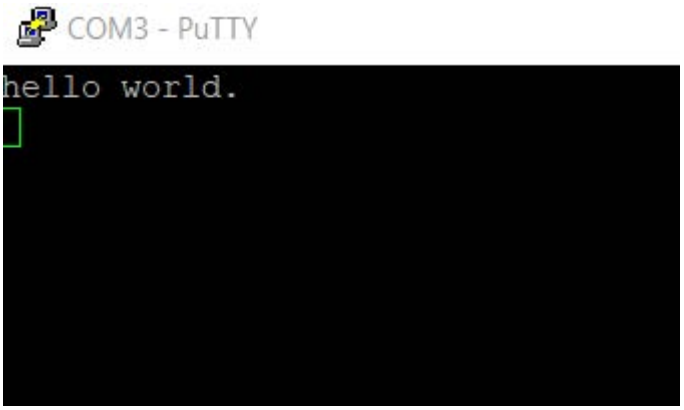
3. In IAR, click the **Download and Debug** button to download the application to the target.

4. The application is then downloaded to the target and automatically runs to the main() function.



5. Run the code by clicking the **Go** button.

6. The hello_world application is now running and a banner is displayed on the terminal. If this is not true, check your terminal settings and connections.



Parent topic:*Running a Demo Application using IAR*

Run a demo using MCUXpresso IDE

Note: Ensure that the MCUXpresso IDE toolchain is included when generating the MCUXpresso SDK package.

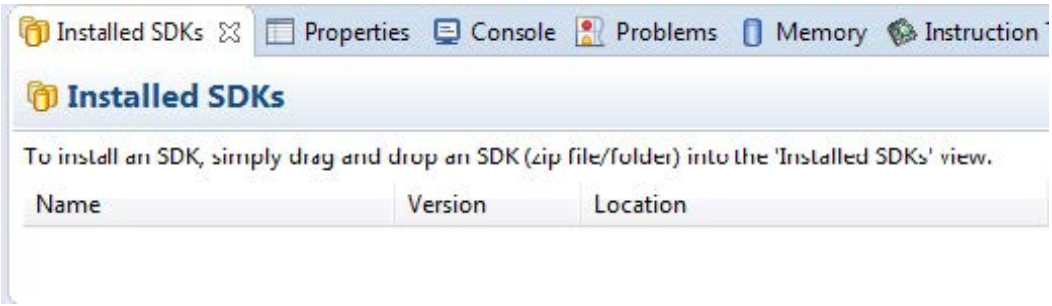
This section describes the steps required to configure MCUXpresso IDE to build, run, and debug example applications. The hello_world demo application targeted for the KW45B41Z-EVK hardware platform is used as an example, though these steps can be applied to any example application in the MCUXpresso SDK.

Select the workspace location Every time MCUXpresso IDE launches, it prompts the user to select a workspace location. MCUXpresso IDE is built on top of Eclipse which uses workspace to store information about its current configuration, and in some use cases, source files for the projects are in the workspace. The location of the workspace can be anywhere, but it is recommended that the workspace be located outside of the MCUXpresso SDK tree.

Parent topic:*Run a demo using MCUXpresso IDE*

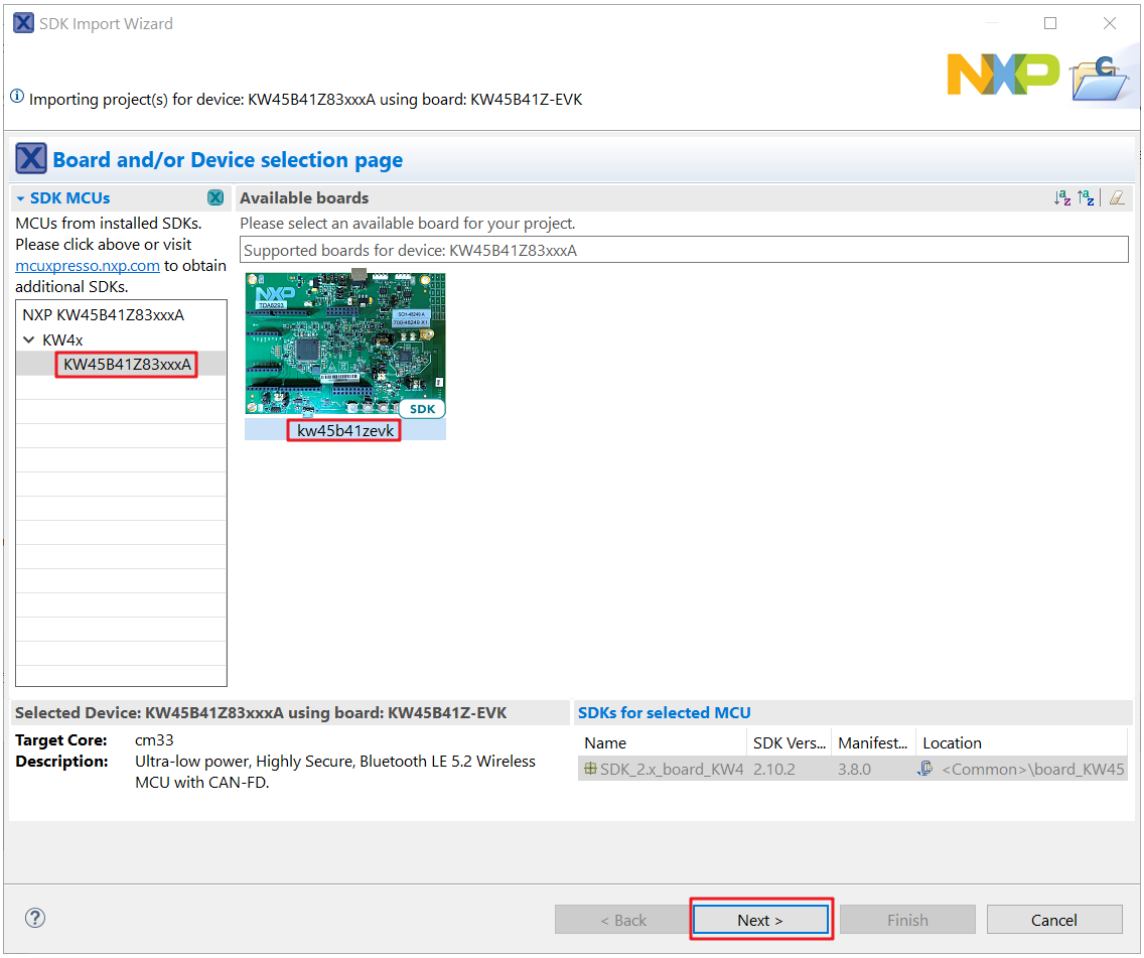
Build an example application To build an example application, follow these steps.

1. Drag and drop the SDK zip file into the **Installed SDKs** view to install an SDK. In the window that appears, click **OK** and wait until the import has finished.

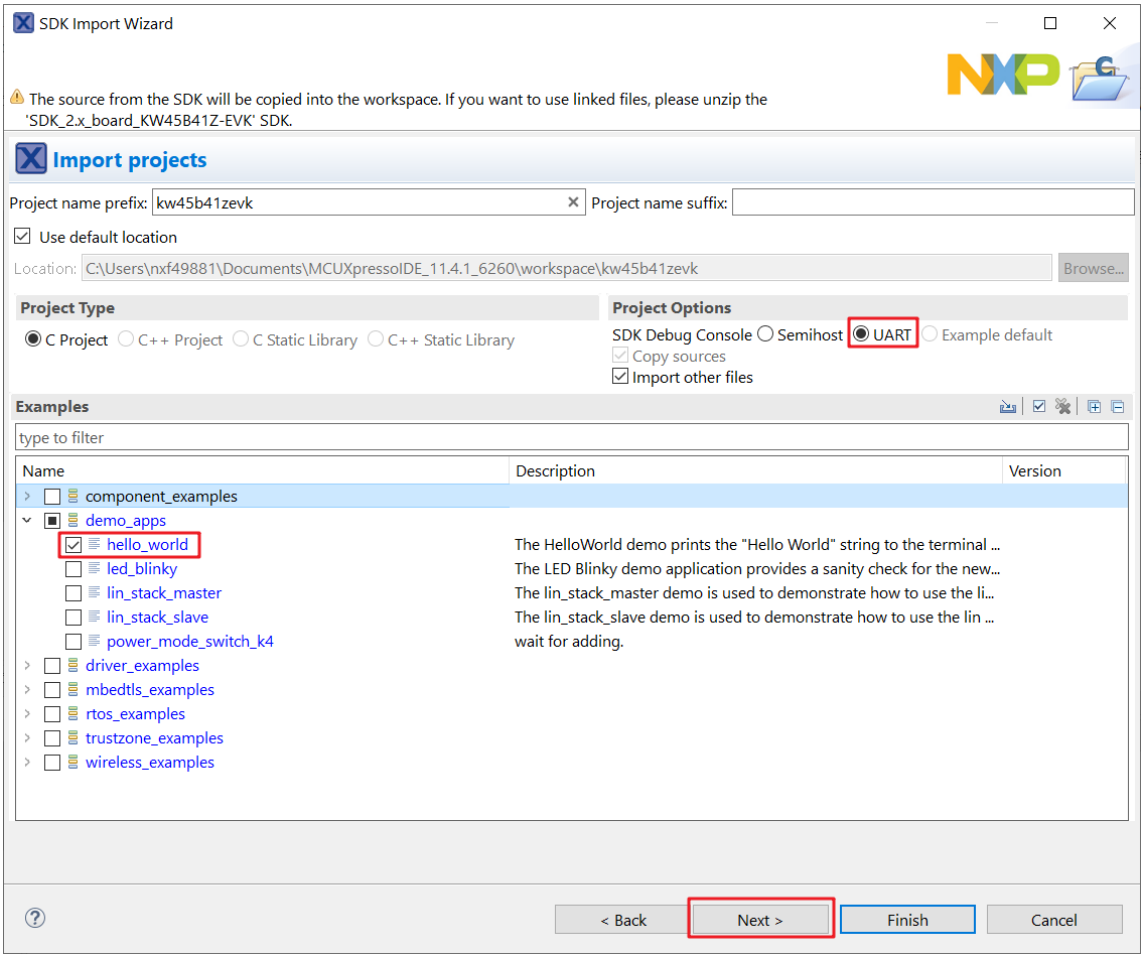


2. On the **Quickstart Panel**, click **Import SDK example(s)...**.

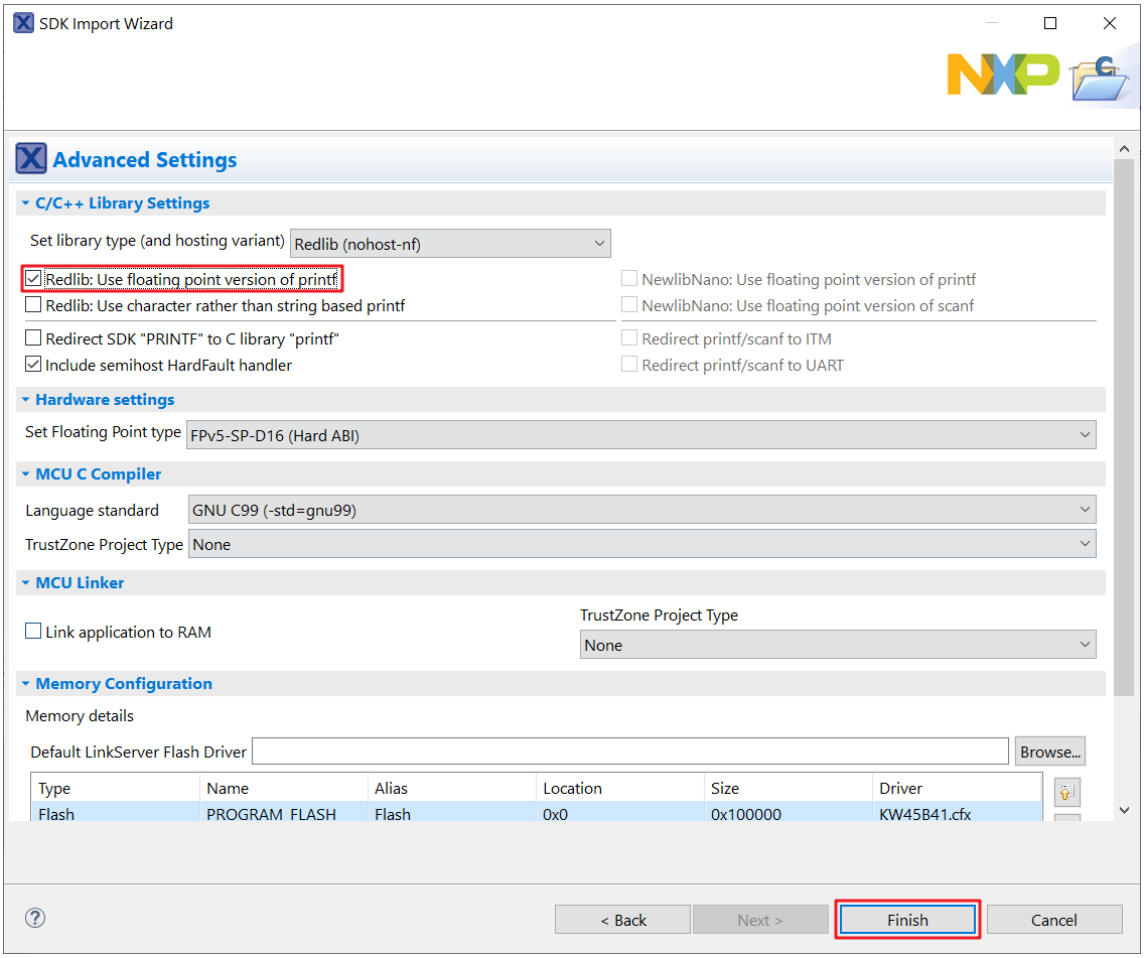
3. In the window that appears, expand the **KW4x** folder and select **KW45B41Z83xxxA**. Then, select **kw45b41zevk** and click **Next**.



4. Expand the demo_apps folder and select hello_world. Then, click **Next**.



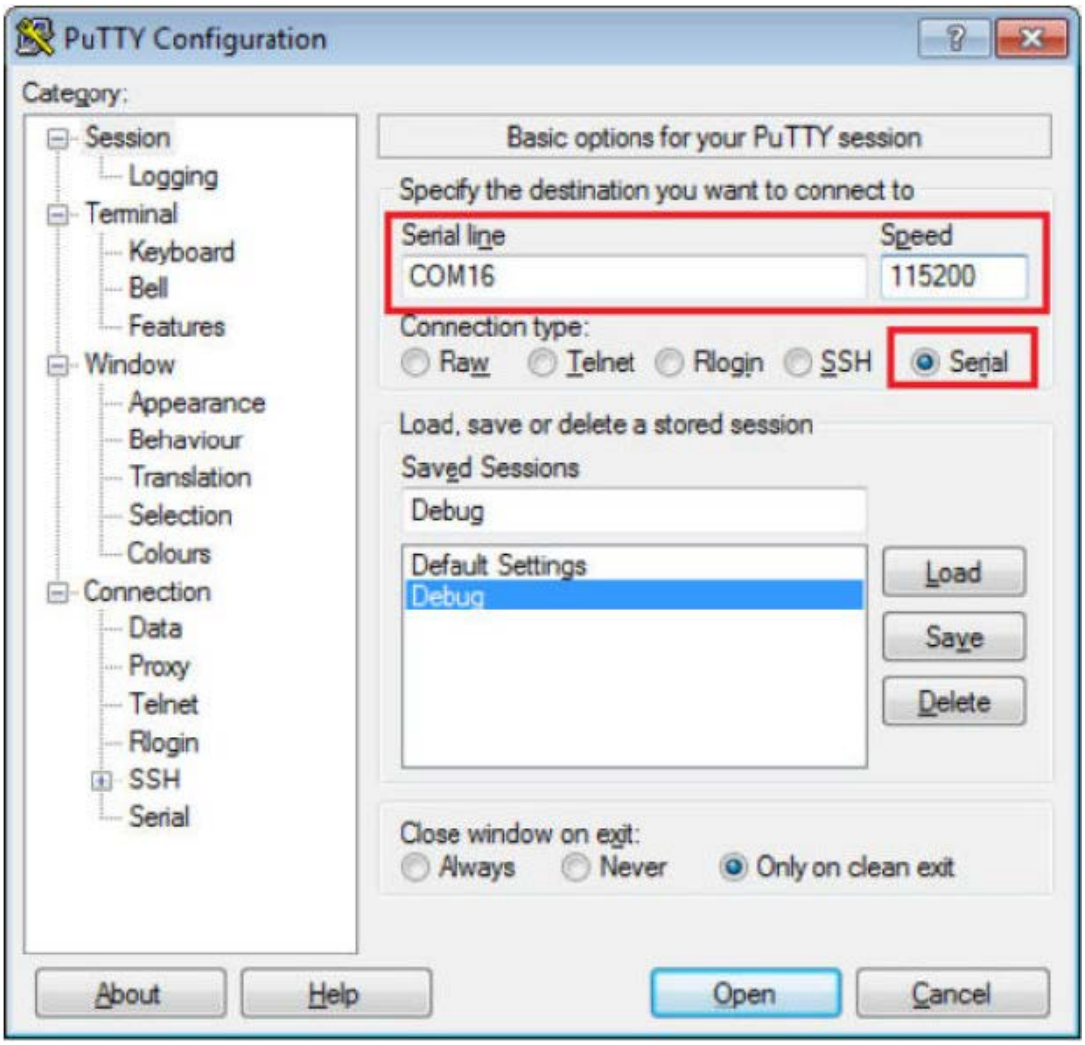
5. Ensure **Redlib: Use floating point version of printf** is selected if the example prints floating point numbers on the terminalfor demo applications such as adc_basic, adc_burst, adc_dma, and adc_interrupt. Otherwise, it is not necessary to select this option. Then, click **Finish**.



Parent topic:*Run a demo using MCUXpresso IDE*

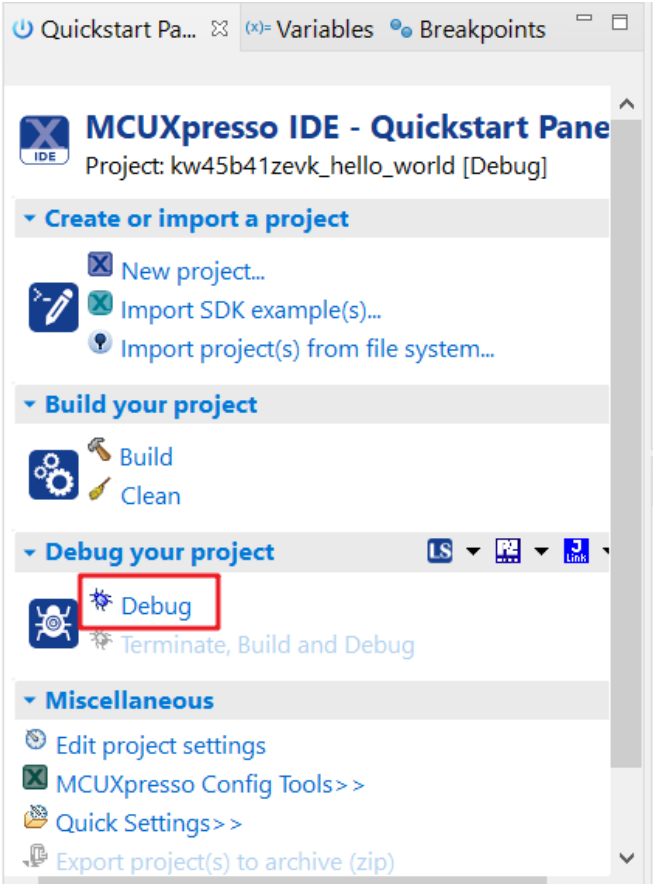
Run an example application To download and run the application, perform the following steps:

1. Connect the development platform to your PC via USB cable between the USB connector (J14) and the PC USB connector.
2. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM port number, see *How to determine COM Port*). Configure the terminal with these settings:
   1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in board.h file)
   2. No parity
   3. 8 data bits

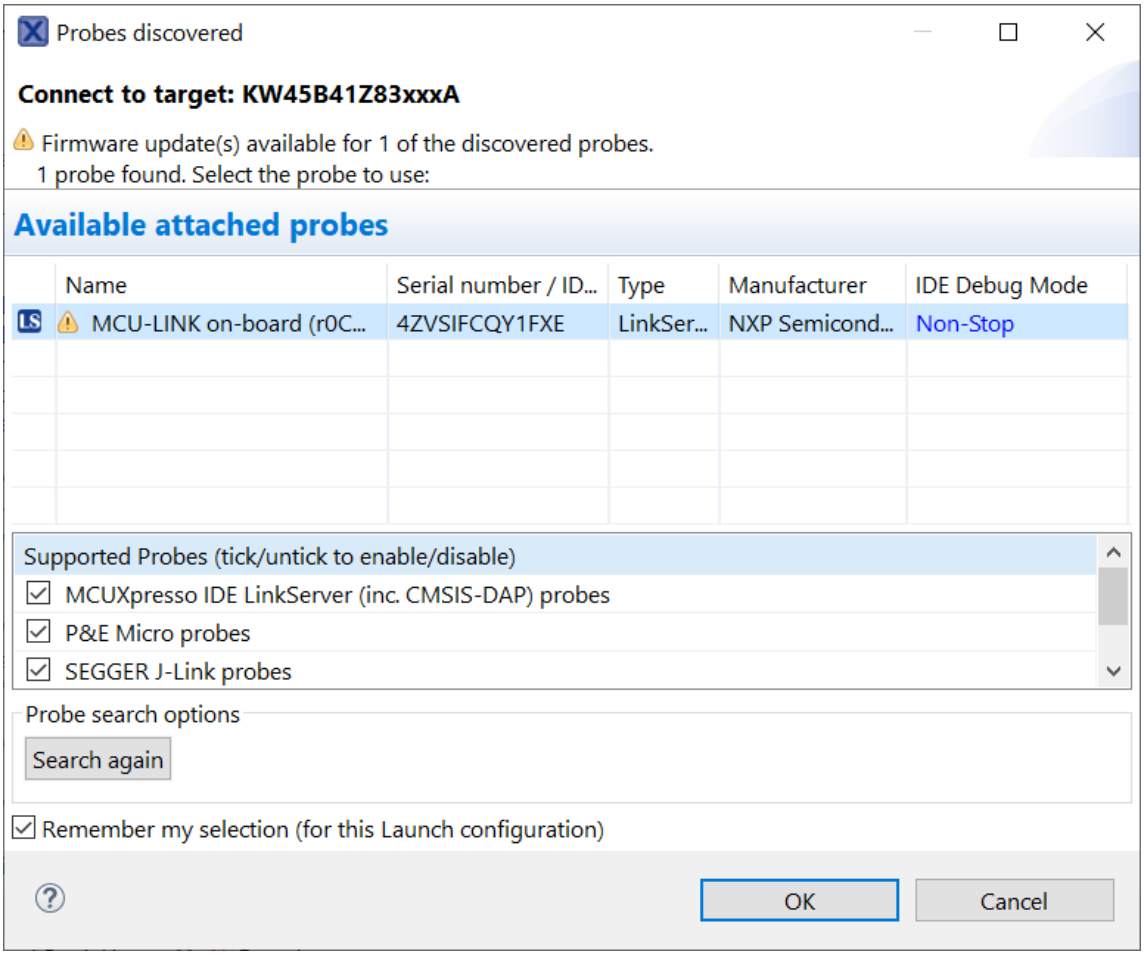


4. 1 stop bit

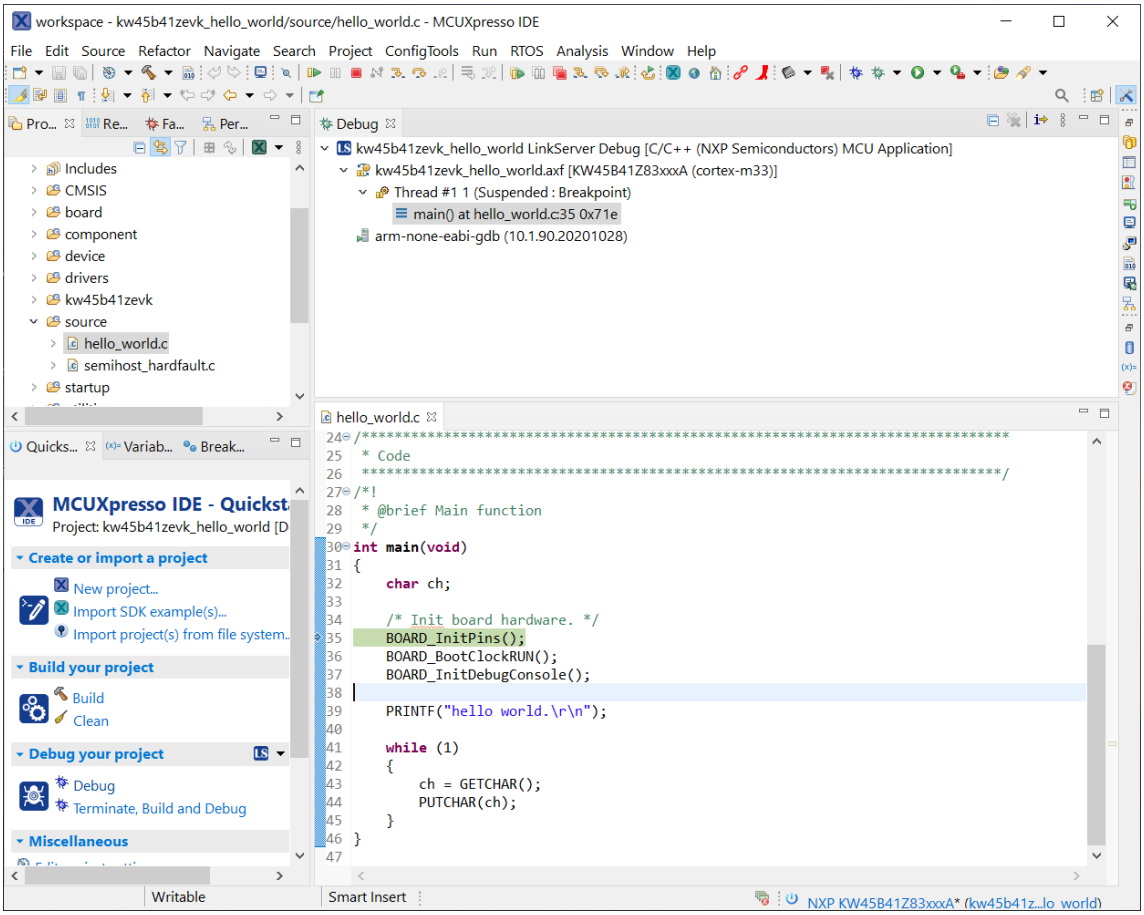
3. On the **Quickstart Panel**, click on **Debug**.



4. The first time you debug a project, the **Debug Emulator Selection** dialog is displayed, showing all supported probes that are attached to your computer. Select the probe through which you want to debug and click **OK**. (For any future debug sessions, the stored probe selection is automatically used, unless the probe cannot be found.)

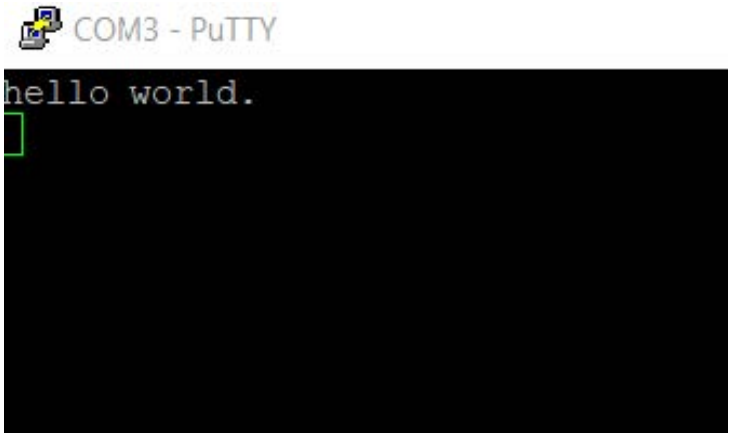


5. The application is downloaded to the target and automatically runs to `main()`.



6. Start the application by clicking **Resume**.

The hello_world application is now running and a banner is displayed on the terminal. If this is not the case, check your terminal settings and connections.

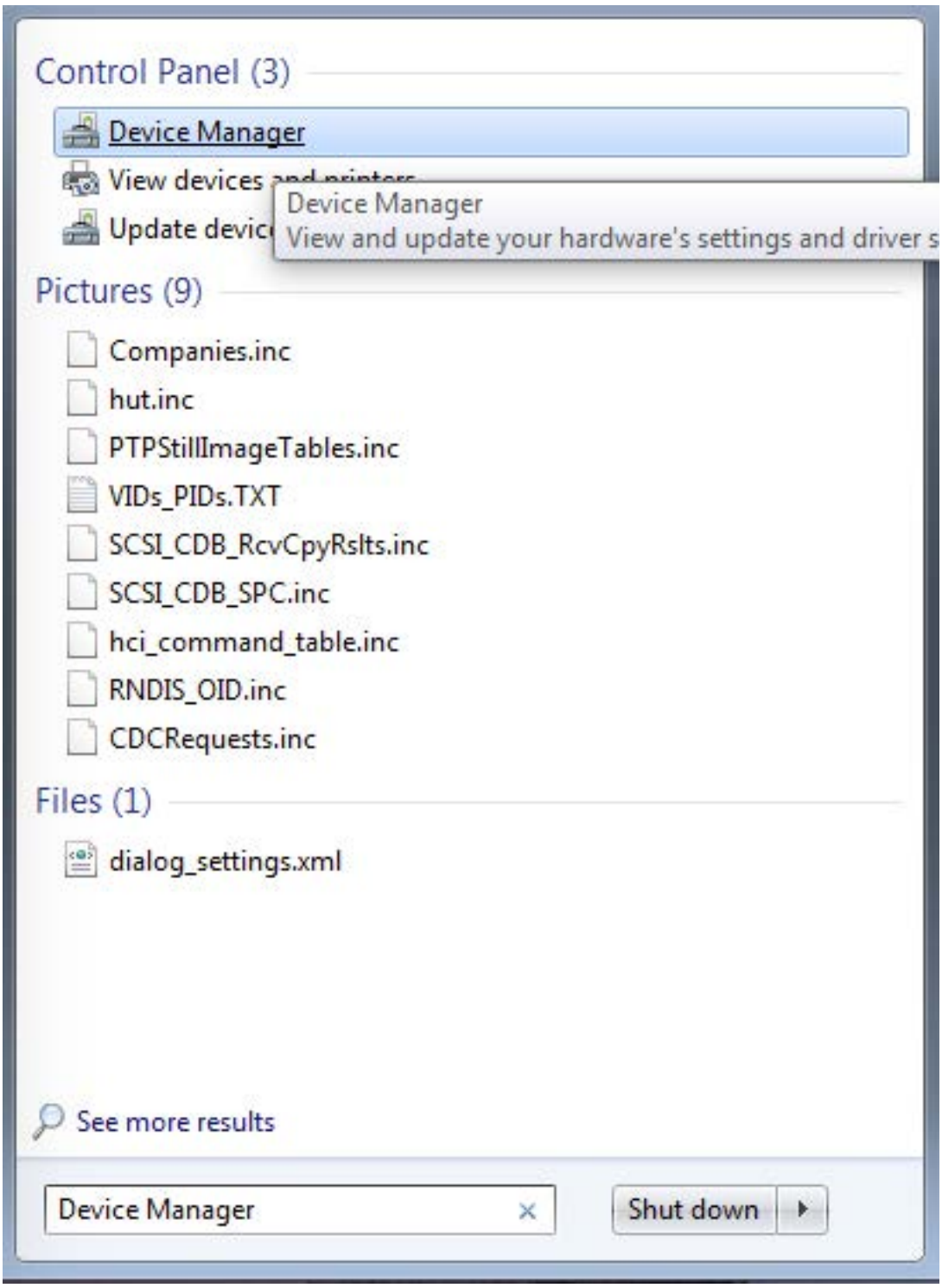


Parent topic:*Run a demo using MCUXpresso IDE*

How to determine COM Port

This section describes the steps necessary to determine the debug COM port number of your NXP hardware development platform. All NXP boards ship with a factory programmed, on-board debug interface MCU-LINK.

1. To determine the COM port, open the Windows operating system **Device Manager**. This can be achieved by going to the Windows operating system **Start** menu and typing **Device Manager** in the search bar, as shown in *Figure 1*.



2. In the **Device Manager**, expand the **Ports (COM & LPT)** section to view the available ports.

How to set the board to the Bootloader ISP mode

On the KW45B41Z-EVK board:

1. Press and hold the BOOT CONFIG switch, **SW4**.
2. Connect the KW45B41Z-EVK board via the micro-USB connector to the PC.
3. Release the switch **SW4**.
4. Check the associated USB port number on the PC (such as, COM10).
5. While the KW45B41Z is in bootloader ISP mode, navigate to the folder where *blhost.exe* is located.
6. Type the command `.\blhost.exe -p COM10 -- get-property 1` to make sure that the KW45B41Z is in Bootloader ISP mode, you should see:

   Ping responded in 1 attempt(s)

   Inject command 'get-property'

   Response status = 0 (0x0) Success.

   Response word 1 = 1258488064 (0x4b030100)

   Current Version = K3.1.0

On a custom board:

1. Short the BOOT_CFG pin to VDD while reset.

Updating debugger firmware

The KW45B41Z-EVK board comes with a CMSIS-DAP-compatible debug interface (known as MCU-Link). This firmware in this debug interface may be updated using the host computer scripts. This firmware is typically used when switching between the default debugger protocol (CMSIS-DAP) to SEGGER J-Link, or for updating this firmware with new releases of these. This section contains the steps to re-program the debug probe firmware.

NXP provides the MCU-Link utility, which is the recommended tool for programming the latest versions of CMSIS-DAP and J-Link firmware onto MCU-Link. The utility can be downloaded from https://www.nxp.com/design/microcontrollers-developer-resources/mcu-link-debug-probe:MCU-LINK.

These steps show how to update the debugger firmware on your board for Windows operating system. For Linux OS, follow the instructions described in MCU-Link user guide, https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools-/mcu-link-debug-probe:MCU-LINK.

1. Install the MCU-Link utility.
2. Unplug the board's USB cable.
3. Install the jumper on JP20.
4. Connect the probe to the host via USB (use Link USB connector).
5. Open a command shell and call the appropriate script located in the MCU-Link installation directory, *<MCU-Link install dir>*.
   1. To program CMSIS-DAP debug firmware: *<MCU-Link install dir>/scripts/program_CMSIS*.
   2. To program J-Link debug firmware: *<MCU-Link install dir>/scripts/program_JLINK*.
6. Remove the jumper on JP20.
7. Re-power the board by removing the USB cable and plugging it in again.

Updating NBU for Wireless Examples

Alert code:

Here you will find two type of images for the NBU FW:

1. SB3 File Type
   - This is for EVK users only. The EVK's come programmed with set keys for ease of use in development
2. XIP File Type
   - This is for samples that are not programmed with any keys. For these devices, you can create your custom keys, then create an SB3 file based on this XIP image

Name	Date modified	Type	Size
Today			
kw45b41_nbu_ble_hosted.sb3	5/14/2025 9:31 AM	SB3 File	206 KB
kw45b41_nbu_ble_hosted.xip	5/14/2025 9:31 AM	XIP File	171 KB

To update the NBU, you may use the SPSDK command line tool.

1. Open the path to your SPSDK folder and activate the virtual environment
   - >> venv\Scripts\activate
2. Place your device in ISP mode. For this example we will use the UART peripheral by connecting a USB cable to J14. On the EVK you can enter ISP by the following method
   - Make sure a jumper is on JP25
   - Press and hold SW4, press and release Reset, and then release SW4
3. Once the device is connected you may check your devices available using SPSDK to find the COM port it is connected to.

- >> nxpdevscan

```
(venv) C:\spsdk>nxpdevscan
-------- Connected NXP USB Devices --------

MCU-LINK on-board (r0C3) CMSIS-DAP V2.250 - NXP Semiconductors
Vendor ID: 0x1fc9
Product ID: 0x0143
Path: HID\VID_1FC9&PID_0143&MI_00\7&7A58EDD&0&0000
Path Hash: 5e2b8187
Name:
Serial number: CWBDF443XGNEL

MCU-LINK NXP TRACE/POWER - NXP Semiconductors
Vendor ID: 0x1fc9
Product ID: 0x0143
Path: HID\VID_1FC9&PID_0143&MI_01\7&2B7CCC9F&0&0000
Path Hash: 6d0c0bef
Name:
Serial number: CWBDF443XGNEL

-------- Connected NXP UART Devices --------

Port: COM27
Type: mboot device

-------- Connected NXP SIO Devices --------
```

4. Then you may run the 'receive-sb-file' command to load the sb3 file.
   - >> blhost -p COM27 receive-sb-file path_to_SDK\kw45b41_nbu_ble_hosted_a1.sb3

1.3 Getting Started with MCUXpresso SDK GitHub

1.3.1 Getting Started with MCUXpresso SDK Repository

Welcome to the **GitHub Repository SDK Guide**. This documentation provides instructions for setting up and working with the MCUXpresso SDK distributed in a **multi-repository model**. The SDK is distributed across multiple GitHub repositories and managed using the **Zephyr West** tool, enabling modular development and streamlined workflows.

Overview

The GitHub Repository SDK approach offers:

- **Modular Structure**: Multiple repositories for flexibility and scalability.
- **Zephyr West Integration**: Simplified repository management and synchronization.
- **Cross-Platform Support**: Designed for MCUXpresso SDK development environments.

Benefits of the Multi-Repository Approach

- **Scalability**: Easily add or update components without impacting the entire SDK.
- **Collaboration**: Enables distributed development across teams and repositories.

- **Version Control**: Independent versioning for components ensures better stability.
- **Automation**: Zephyr West simplifies dependency handling and repository synchronization.

Setup and Configuration

Follow these steps to prepare your development environment:

Development Tools Installation This guide explains how to install the essential tools for development with the MCUXpresso SDK.

Quick Start: Automated Installation (Recommended) The **MCUXpresso Installer** is the fastest way to get started. It automatically installs all the basic tools you need.

1. **Download the MCUXpresso Installer** from: Dependency-Installation
2. **Run the installer** and select **"MCUXpresso SDK Developer"** from the menu
3. **Click Install** and let it handle everything automatically

Manual Installation If you prefer to install tools manually or need specific versions, follow these steps:

Essential Tools

Git - Version Control **What it does**: Manages code versions and downloads SDK repositories from GitHub.

Installation:

- Visit git-scm.com
- Download for your operating system
- Run installer with default settings
- **Important**: Make sure "Add Git to PATH" is selected during installation

Setup:

```
git config --global user.name "Your Name"
git config --global user.email "youremail@example.com"
```

Python - Scripting Environment **What it does**: Runs build scripts and SDK tools.

Installation:

- Install Python **3.10 or newer** from python.org
- **Important**: Check "Add Python to PATH" during installation

West - SDK Management Tool **What it does**: Manages SDK repositories and provides build commands. The west tool is developed by the Zephyr project for managing multiple repositories.

Installation:

```
pip install -U west
```

Minimum version: 1.2.0 or newer

Build System Tools

CMake - Build Configuration **What it does**: Configures how your projects are built.

Recommended version: 3.30.0 or newer

Installation:

- **Windows**: Download `.msi` installer from cmake.org/download
- **Linux**: Use package manager or download from cmake.org
- **macOS**: Use Homebrew (`brew install cmake`) or download from cmake.org

Ninja - Fast Build System **What it does**: Compiles your code quickly.

Minimum version: 1.12.1 or newer

Installation:

- **Windows**: Usually included, or download from ninja-build.org
- **Linux**: `sudo apt install ninja-build` or download binary
- **macOS**: `brew install ninja` or download binary

Ruby - IDE Project Generation (Optional) **What it does**: Generates project files for IDEs like IAR and Keil.

When needed: Only if you want to use traditional IDEs instead of VS Code.

Installation: Follow the Ruby environment setup guide

Compiler Toolchains Choose and install the compiler toolchain you want to use:

Toolchain	Best For	Download Link	Environment Variable
ARM GCC (Recommended)	Most users, free	ARM GNU Toolchain	`ARMGCC_DIR`
IAR EWARM	Professional development	IAR Systems	`IAR_DIR`
Keil MDK	ARM ecosystem	ARM Developer	`MDK_DIR`
ARM Compiler	Advanced optimization	ARM Developer	`ARMCLANG_DIR`

Setting Up Environment Variables After toolchain installation, set an environment variable so the build system locates it:

Windows:

```
# Example for ARM GCC installed in C:\armgcc
setx ARMGCC_DIR "C:\armgcc"
```

Linux/macOS:

```
# Add to ~/.bashrc or ~/.zshrc
export ARMGCC_DIR="/usr"  # or your installation path
```

Verify Your Installation After installation, verify everything works by opening a terminal/command prompt and running these commands:

```
# Check each tool - you should see version numbers
git --version
python --version
west --version
cmake --version
ninja --version
arm-none-eabi-gcc --version  # (if using ARM GCC)
```

Troubleshooting Installation Issues **"Command not found" errors**:

- The tool isn't in your system PATH
- **Solution**: Add the installation directory to your PATH environment variable

Python/pip issues:

- Try using `python3` and `pip3` instead of `python` and `pip`
- On Windows, run the Command Prompt as an Administrator

Slow downloads:

- Add timeout option: `pip install -U west --default-timeout=1000`
- Use alternative mirror: `pip install -U west -i https://pypi.tuna.tsinghua.edu.cn/simple`

GitHub Repository Setup This guide explains how to initialize your MCUXpresso SDK workspace from GitHub repositories using the west tool. The GitHub Repository SDK uses multiple repositories hosted on GitHub to provide modular, flexible development.

Prerequisites Verify the requirements:

System Requirements:

- Python 3.8 or later
- Git 2.25 or later
- CMake 3.20 or later
- Build tools for your target platform

Verification Commands:

```
python --version    # Should show 3.8+
git --version       # Should show 2.25+
cmake --version     # Should show 3.20+
west --version      # Should show west tool installation
```

Workspace Initialization The GitHub Repository SDK uses the Zephyr west tool to manage multiple repositories containing different SDK components.

Step 1: Initialize Workspace Create and initialize your SDK workspace from GitHub:

Get the latest SDK from main branch:

```
west init -m https://github.com/nxp-mcuxpresso/mcuxsdk-manifests.git mcuxpresso-sdk
```

Get SDK at specific revision:

```
west init -m https://github.com/nxp-mcuxpresso/mcuxsdk-manifests.git mcuxpresso-sdk --mr {revision}
```

Note: Replace {revision} *with the desired release tag, such as* v25.09.00

Step 2: Choose Your Repository Update Strategy Navigate to the SDK workspace:

```
cd mcuxpresso-sdk
```

The west tool manages multiple GitHub repositories containing different SDK components. You have two options for downloading:

Option A: Download All Repositories (Complete SDK) Download all SDK repositories for comprehensive development:

```
west update
```

This command downloads all the repositories defined in the manifest from GitHub. Initial download takes several minutes and requires ~7 GB of disk space.

Best for:

- Exploring the complete SDK
- Multi-board development projects
- Comprehensive middleware evaluation

Option B: Targeted Repository Download (Recommended) Download only repositories needed for your specific board or device to save time and disk space:

```
# For specific board development
west update_board --set board your_board_name

# For specific device family development
west update_board --set device your_device_name

# List available repositories before downloading
west update_board --set board your_board_name --list-repo
```

Best for:

- Single board development

- Faster setup and reduced disk usage
- Focused development workflows

Examples:

```
# Update only repositories for FRDM-MCXW23 board
west update_board --set board frdmmcxw23

# Update only repositories for MCXW23 device family
west update_board --set device mcxw23
```

Step 3: Verify Installation Confirm successful setup:

```
# Verify workspace structure
ls -la
# Should show: manifests/ and mcuxsdk/ directories

# Test build system
west list_project -p examples/demo_apps/hello_world
# Should display available build configurations
```

Advanced Repository Management The west extension command `update_board` provides advanced repository management capabilities for optimized workspace setup with GitHub repositories.

Board-Specific Setup Update only repositories required for a specific board:

```
# Update only repositories for specific board, e.g., frdmmcxw23
west update_board --set board frdmmcxw23

# List available repositories for the board before updating
west update_board --set board frdmmcxw23 --list-repo
```

Device-Specific Setup Update only repositories required for a specific device family:

```
# Update only repositories for specific device, e.g., MCXW235
west update_board --set device mcxw23

# List available repositories for the device family
west update_board --set device mcxw23 --list-repo
```

Custom Configuration For advanced users who want to create custom repository combinations:

```
# Use custom configuration file
west update_board --set custom path/to/custom-config.yml

# Generate custom configuration template
cp manifests/boards/custom.yml.template my-custom-config.yml
```

Benefits of Targeted Setup **Reduced Download Size**

- Download only components needed for your target board or device
- Significantly faster initial setup for focused development

- Typical reduction from 7 GB to 2GB

Optimized Workspace

- Cleaner workspace with relevant components only
- Reduced disk space usage
- Faster repository operations

Flexible Development

- Switch between different board configurations easily
- Maintain separate workspaces for different projects
- Include optional components as needed

Repository Information Before setting up your workspace, you can explore what repositories are available:

```
# Display repository information in console
west update_board --set board frdmmcxw23 --list-repo

# Export repository information to YAML file for reference
west update_board --set board frdmmcxw23 --list-repo -o board-repos.yml
```

This command lists all the available repositories with descriptions and outlines the included components in the workspace.

Package Generation (Optional) The `update_board` command can also generate ZIP packages for offline distribution:

```
# Generate board-specific SDK package
west update_board --set board frdmmcxw23 -o frdmmcxw23-sdk.zip
```

Note: Package generation is primarily intended for creating custom SDK distributions. For regular development, use the workspace update commands without the `-o` option.

Workspace Management

Updating Your Workspace Keep your SDK current with latest updates from GitHub:

For Complete SDK Workspace:

```
# Update manifest repository
cd manifests
git pull

# Update all component repositories
cd ..
west update
```

For Targeted Workspace:

```
# Update manifest repository
cd manifests
git pull

# Update board-specific repositories
cd ..
west update_board --set board your_board_name
```

Workspace Status Check workspace synchronization status:

```
# Show status of all repositories
west status

# Show detailed information about repositories
west list
```

Troubleshooting **Network Issues:**

- Use `west update --keep-descendants` for partial failures
- Configure Git credentials for private repositories
- Check firewall settings for Git protocol access

Permission Issues:

- Ensure write permissions in workspace directory
- Run commands without sudo/administrator privileges
- Verify Git SSH key configuration for authenticated access

Disk Space:

- Full SDK workspace requires approximately 7-8 GB
- Targeted workspace typically requires 1-2 GB
- Use board-specific setup to reduce workspace size

Repository Management Issues:

- Verify board/device names match available configurations
- Check that custom YAML files follow the correct template format
- Use `--list-repo` to verify available repositories before setup

Next Steps With your workspace initialized:

1. Review *Workspace Structure* to understand the layout
2. Build your first project with *First Build Guide*
3. Explore *Development Workflows MCUXPresso VSCode* or *Development Workflows Command Line* for the details on project setup and execution

For advanced repository management, see the west tool documentation.

Explore SDK Structure and Content

Learn about the organization of the SDK and its components:

SDK Architecture Overview The MCUXpresso SDK uses a modular architecture where software components are distributed across multiple repositories hosted on GitHub and managed through the west tool. This approach provides flexibility, maintainability, and enables selective component inclusion.

Repository Organization Based on the manifest structure, the SDK consists of four main repository categories:

Manifest Repository The manifest repo (mcuxsdk-manifests) contains the west.yml manifest file that tracks all other repositories in the SDK.

Base Repositories Recorded in `submanifests/base.yml` and loaded in the root west.yml manifest file. These are the foundational repositories that build the SDK:

- **Devices**: MCU-specific support packages
- **Examples**: Demonstration applications and code samples
- **Boards**: Board support packages

Middleware Repositories Recorded in the `submanifests/middleware` subdirectory, categorized according to functionality:

- **Connectivity**: Networking stacks, USB, and communication protocols
- **Security**: Cryptographic libraries and secure boot components
- **Wireless**: Bluetooth, IEEE 802.15.4, and other wireless protocols
- **Graphics**: Display drivers and UI frameworks
- **Audio**: Audio processing and voice recognition libraries
- **Machine Learning**: AI inference engines and neural network libraries
- **Safety**: IEC60730B safety libraries
- **Motor Control**: Motor control and real-time control libraries

Internal Repositories Recorded in `submanifests/internal.yml` and grouped into the "bifrost" group. These are only visible to NXP internal developers and hosted on NXP internal git servers.

Repository Hosting Public repositories are hosted on GitHub under these organizations:

- nxp-mcuxpresso
- NXP
- nxp-zephyr

Internal repositories are hosted on NXP's private Git infrastructure.

Benefits of This Architecture **Selective Integration**: Projects include only required components, reducing memory footprint and build complexity.

Independent Versioning: Each component maintains its own release cycle and version control.

Community Collaboration: Public repositories accept community contributions through standard Git workflows.

Scalable Maintenance: Component owners can update their repositories without affecting the entire SDK.

Workspace Management The west tool manages repository synchronization, version tracking, and workspace updates. All repositories are checked out under the `mcuxsdk/` directory with their designated paths defined in the manifest files.

Workspace Structure After you initialize your SDK workspace, it creates a specific directory structure that organizes all SDK components. This structure is identical for both GitHub Repository SDK and Repository-Layout SDK Package.

Top-Level Organization

```
your-sdk-workspace/
  manifests/        # West manifest repository
  mcuxsdk/          # Main SDK content
```

The mcuxsdk/ directory serves as your primary working directory and contains all the SDK components.

SDK Component Layout Based on the actual SDK structure, the main directories include:

Directory	Contents	Purpose
arch/	Architecture-specific files	ARM CMSIS, build configurations
cmake	Build system modules	CMake configuration and build rules
compo	Software components	Reusable software libraries and utilities
devices	Device support packages	MCU-specific headers, startup code, linker scripts
drivers	Peripheral drivers	Hardware abstraction layer for MCU peripherals
examp	Sample applications	Demonstration code and reference implementations
middle	Optional software stacks	Networking, graphics, security, and other libraries
rtos/	Operating system support	FreeRTOS integration
scripts	Build and utility scripts	West extensions and development tools
svd	Svd files for devices, this is optional because of large size. Customers run west manifest config group.filter +optional and west update mcux-soc-svd to get this folder.	

Example Organization Examples follow a two-tier structure separating common code from board-specific implementations:

Common Example Files

```
examples/demo_apps/hello_world/
  CMakeLists.txt      # Build configuration
  example.yml         # Example metadata
  hello_world.c       # Application source code
  Kconfig             # Configuration options
  readme.md           # General documentation
```

Board-Specific Files

```
examples/_boards/your_board/demo_apps/hello_world/
    app.h                    # Board specific application header
    example_board_readme.md  # Board specific documentation
    hardware_init.c          # Board specific hardware initialization
    pin_mux.c                # Pin multiplexing configuration
    pin_mux.h                # Pin multiplexing header definitions
    hello_world.bin          # Pre-built binary for quick testing
    hello_world.mex          # MCUXpresso Config Tools project file
    prj.conf                 # Board specific Kconfig configuration
    reconfig.cmake           # Board specific cmake configuration overrides
```

Device Support Structure Device support is organized hierarchically by MCU family:

```
devices/
    MCX/                 # MCU portfolio
        MCXW/            # MCU family
            MCXW235/     # Specific device
                MCXW235.h          # Device register definitions
                drivers/           # Device-specific drivers
                gcc/               # GNU toolchain files
                iar/               # IAR toolchain files
                mcuxpresso/        # MCUXpresso IDE files
                startup_MCXW235.c # Startup and vector table
                system_MCXW235.c  # System initialization
```

Middleware Organization Middleware components are categorized by functionality and maintained in separate repositories. Based on the manifest files, common middleware categories include:

- **Connectivity**: USB, TCP/IP, industrial protocols
- **Security**: Cryptographic libraries, secure boot
- **Wireless**: Bluetooth, IEEE 802.15.4, Wi-Fi
- **Graphics**: Display drivers, UI frameworks
- **Audio**: Processing libraries, voice recognition
- **Machine Learning**: Inference engines, neural networks
- **Safety**: IEC60730B safety libraries
- **Motor Control**: Motor control and real-time control libraries

Documentation Structure SDK documentation is distributed across multiple locations:

- `docs/` - Core SDK documentation and build infrastructure
- Component repositories - API documentation and integration guides
- Board directories - Hardware-specific setup instructions

For complete documentation, refer to the online documentation.

Understanding Example Structure Each example has **two README files**:

1. **General README:** examples/demo_apps/hello_world/readme.md
   - What the example does
   - General functionality description
   - Common usage information

2. **Board-Specific README:** examples/_boards/{board_name}/demo_apps/hello_world/example_board_readme.md
   - Board-specific setup instructions
   - Hardware connections required
   - Board-specific behavior notes

Tip: Always check both readme files - start with the general one, then read the board-specific one for detailed setup.

Development Workflows

Get started with building and running projects:

Building Your First Project This guide explains how to build and run your first SDK example project using the west build system. This applies to both GitHub Repository SDK and Repository-Layout SDK Package.

Prerequisites

- GitHub Repository SDK workspace initialized OR Repository-Layout SDK Package extracted
- Development board connected via USB
- Build tools installed per *Installation Guide*

Understanding Board Support Use the west extension to discover available examples for your board:

```
west list_project -p examples/demo_apps/hello_world
```

This shows all supported build configurations. You can filter by toolchain:

```
west list_project -p examples/demo_apps/hello_world -t armgcc
```

Basic Build Process

Simple Build Build the hello_world example with default settings:

```
west build -b your_board examples/demo_apps/hello_world
```

The default toolchain is armgcc, and the build system will select the first debug target as default if no config is specified.

Specifying Configuration

```
# Release build
west build -b your_board examples/demo_apps/hello_world --config release

# Debug build (default)
west build -b your_board examples/demo_apps/hello_world --config debug
```

Alternative Toolchains

```
# IAR toolchain
west build -b your_board examples/demo_apps/hello_world --toolchain iar

# Other toolchains as supported by the example
```

Multicore Applications For multicore devices, specify the core ID:

```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config␣
↪flexspi_nor_debug
```

For multicore projects using sysbuild:

```
west build -b evkbmimxrt1170 --sysbuild ./examples/multicore_examples/hello_world/primary -Dcore_
↪id=cm7 --config flexspi_nor_debug --toolchain=armgcc -p always
```

Flash an Application Flash the built application to your board:

```
west flash -r linkserver
```

Debug Start a debug session:

```
west debug -r linkserver
```

Common Build Options

Clean Build Force a complete rebuild:

```
west build -b your_board examples/demo_apps/hello_world -p always
```

Dry Run See the commands that get executed without running them:

```
west build -b your_board examples/demo_apps/hello_world --dry-run
```

Device Variants For boards supporting multiple device variants:

```
west build -b your_board examples/demo_apps/hello_world --device DEVICE_PART_NUMBER --config␣
↪release
```

Project Configuration

CMake Configuration Only Run configuration without building:

```
west build -b your_board examples/demo_apps/hello_world -Dcore_id=cm7 --cmake-only -p
```

Interactive Configuration Launch the configuration GUI:

```
west build -t guiconfig
```

Troubleshooting

Build Failures Use pristine builds to resolve dependency issues:

```
west build -b your_board examples/demo_apps/hello_world -p always
```

Getting Help View the help information for west build:

```
west build -h
```

Check Supported Configurations To see available configuration options and board targets for an example, refer to the below command:

```
west list_project -p examples/demo_apps/hello_world
```

Next Steps

- Explore other examples in the SDK
- Learn about *Command Line Development* for advanced options
- Try *VS Code Development* for integrated development
- Refer *Workspace Structure* to understand the SDK layout

MCUXpresso for VS Code Development This guide covers using MCUXpresso for VS Code extension to build, debug, and develop SDK applications with an integrated development environment.

Prerequisites

- SDK workspace initialized (GitHub Repository SDK or Repository-Layout SDK Package)
- Development tools installed per *Installation Guide*
- Visual Studio Code installed
- MCUXpresso for VS Code extension installed

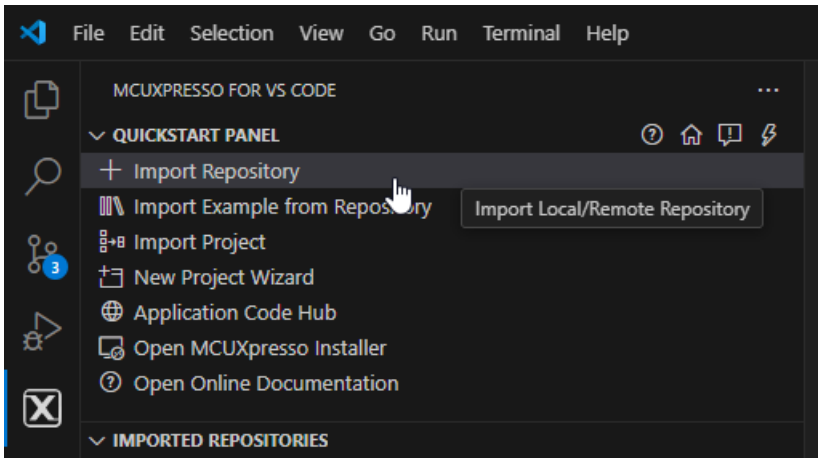
Extension Installation

Install MCUXpresso for VS Code The MCUXpresso for VS Code extension provides integrated development capabilities for MCUXpresso SDK projects. Refer to the MCUXpresso for VS Code Wiki for detailed installation and setup instructions.

SDK Import and Setup

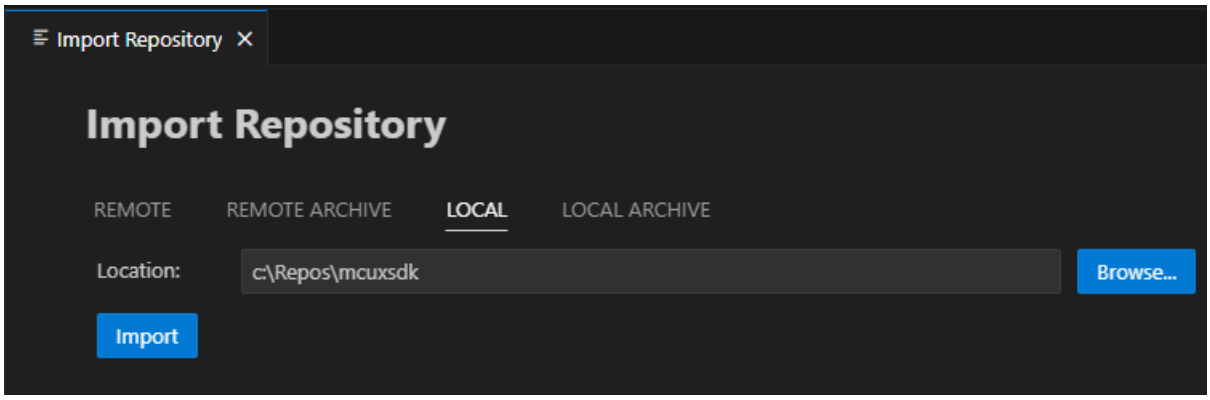
Import Methods The SDK can be imported in several ways. The MCUXpresso for VS Code extension supports both GitHub Repository SDK and Repository-Layout SDK Package distributions.

Import GitHub Repository SDK Click **Import Repository** from the **QUICKSTART PANEL**

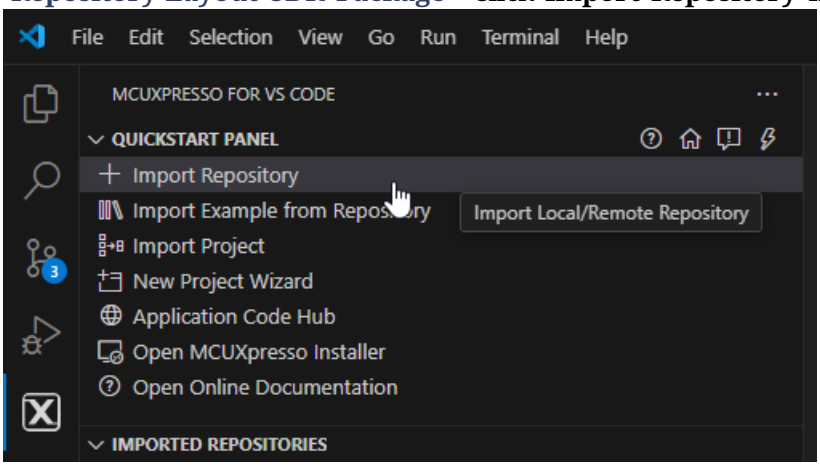


Note: You can import the SDK in several ways. Refer to MCUXpresso for VS Code Wiki for details.

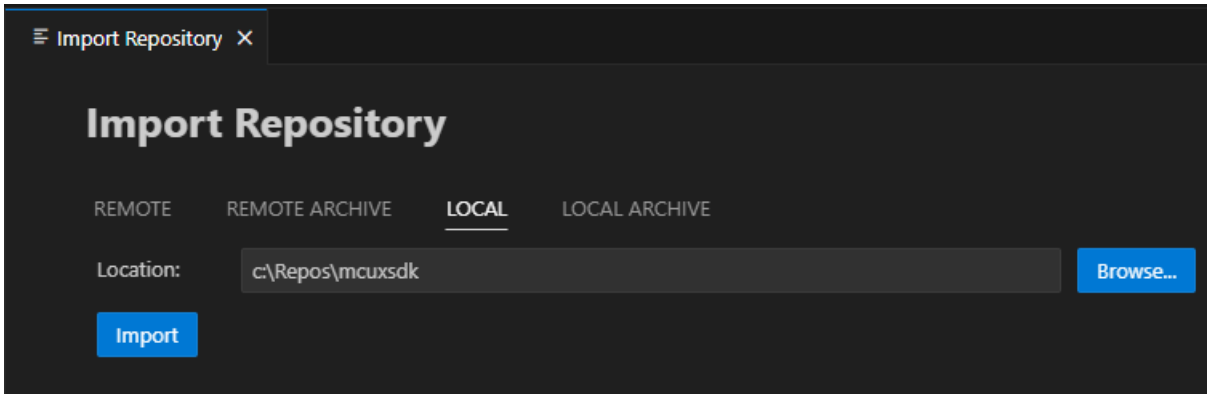
Select **Local** if you've already obtained the SDK according to *setting up the repo*. Select your location and click **Import**.



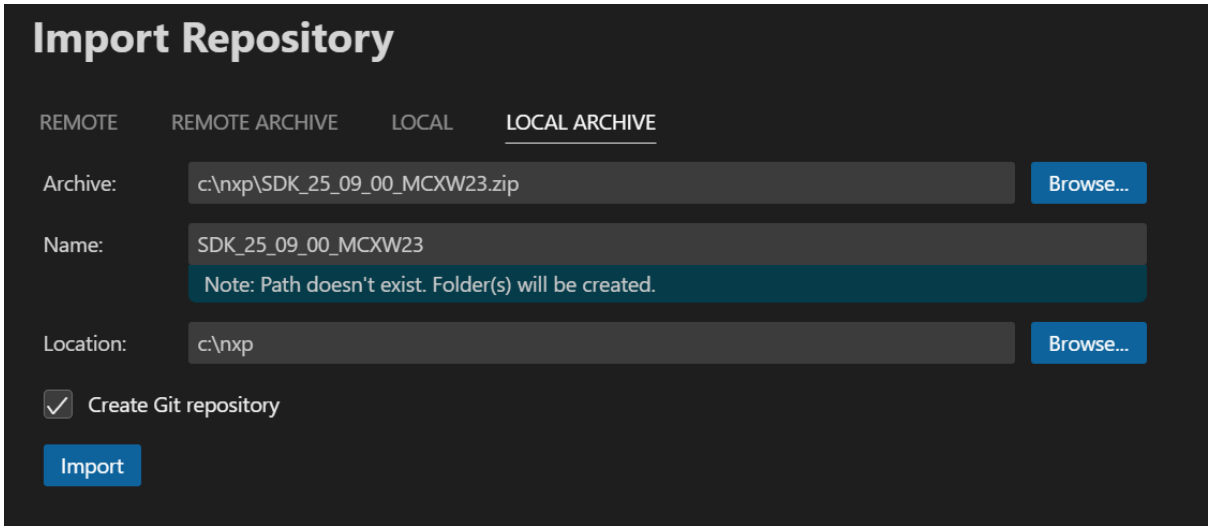
Import Repository-Layout SDK Package Click **Import Repository** from the **QUICKSTART PANEL**



Select **Local** if you've already unzipped the Repository-Layout SDK Package. Select your location and click **Import**.



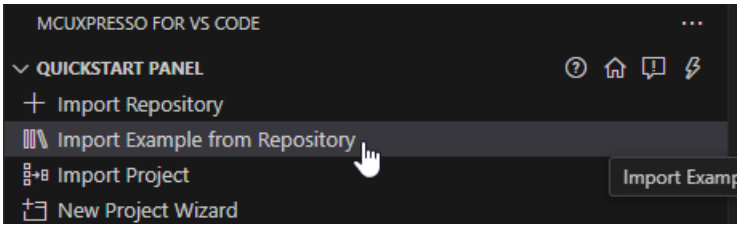
Else if the SDK is ZIP archive, select **Local Archive**, browse to the downloaded SDK ZIP file, fill the link of expect location, then click **Import**.



Building Example Applications

Import Example Project

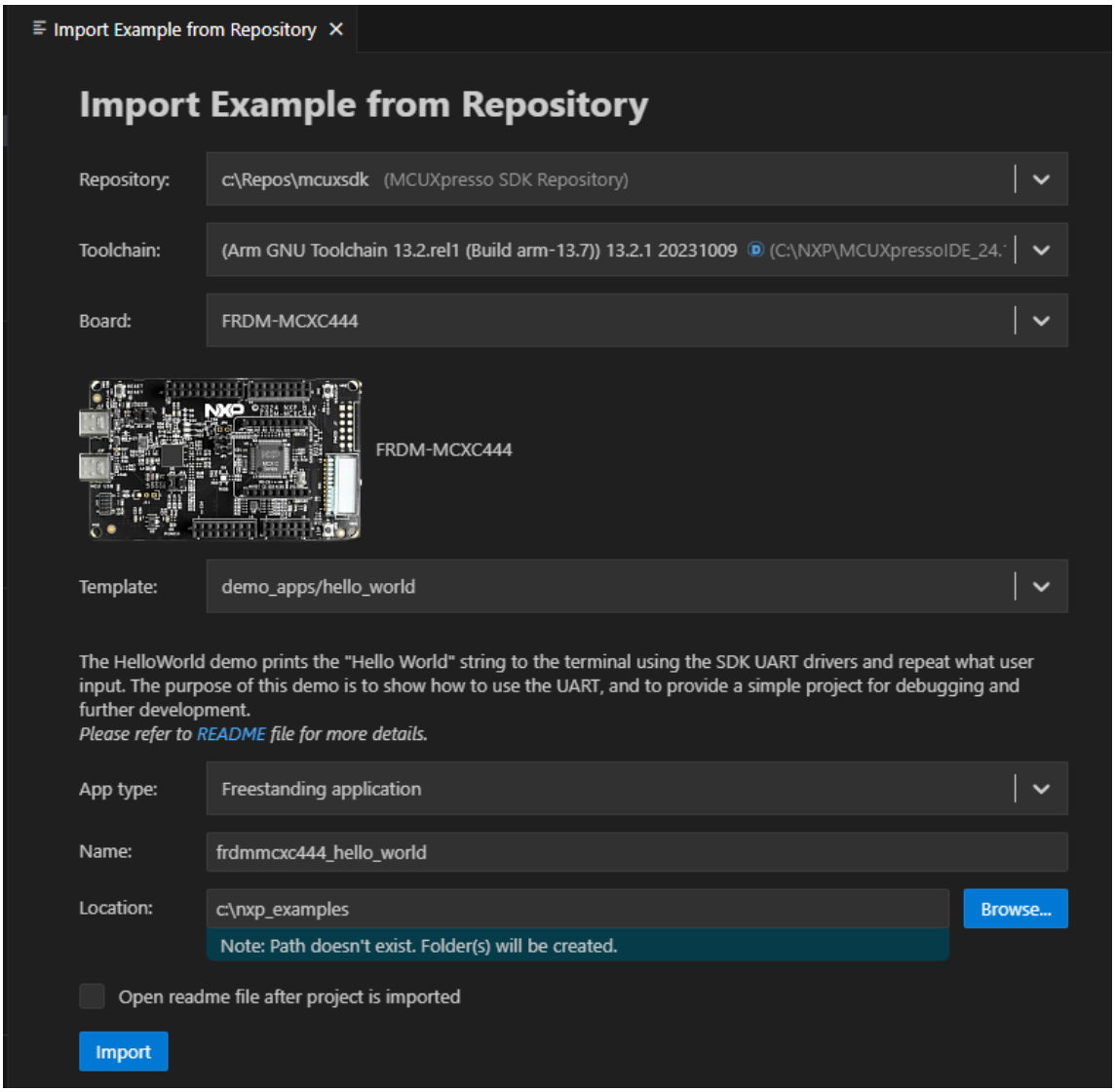
1. Click **Import Example from Repository** from the **QUICKSTART PANEL**



2. Configure project settings:
   - **MCUXpresso SDK**: Select your imported SDK
   - **Arm GNU Toolchain**: Choose toolchain
   - **Board**: Select your target development board
   - **Template**: Choose example category

- **Application**: Select specific example (e.g., hello_world)
- **App type**: Choose between Repository applications or Freestanding applications

3. Click **Import**



Application Types **Repository Applications:**

- Located inside the MCUXpresso SDK
- Integrated with SDK workspace

Freestanding Applications:

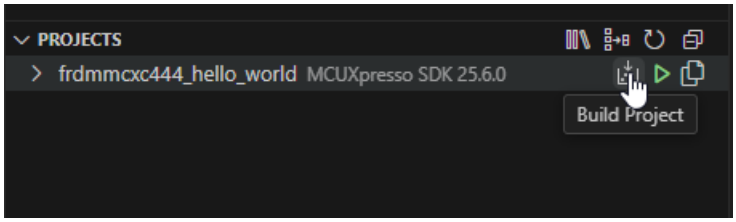
- Imported to user-defined location
- Independent of SDK location

Trust Confirmation VS Code will prompt you to confirm if the imported files are trusted. Click **Yes** to proceed.

Building Projects

Build Process

1. Navigate to **PROJECTS** view
2. Find your project
3. Click the **Build Project** icon

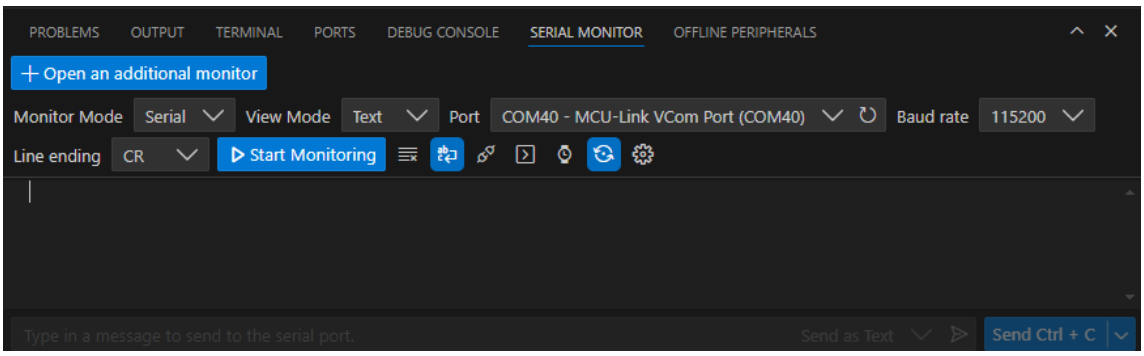


The integrated terminal will display build output at the bottom of the VS Code window.

Running and Debugging

Serial Monitor Setup

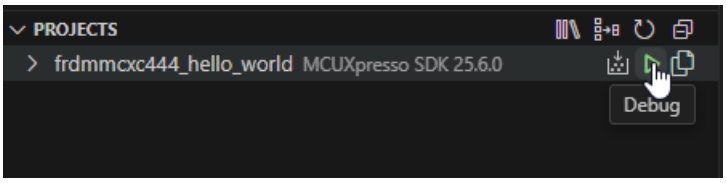
1. Open **Serial Monitor** from VS Code's integrated terminal



2. Configure serial settings:
   - **VCom Port**: Select port for your device
   - **Baud Rate**: Set to 115200

Debug Session

1. Navigate to **PROJECTS** view
2. Click the play button to initiate a debug session

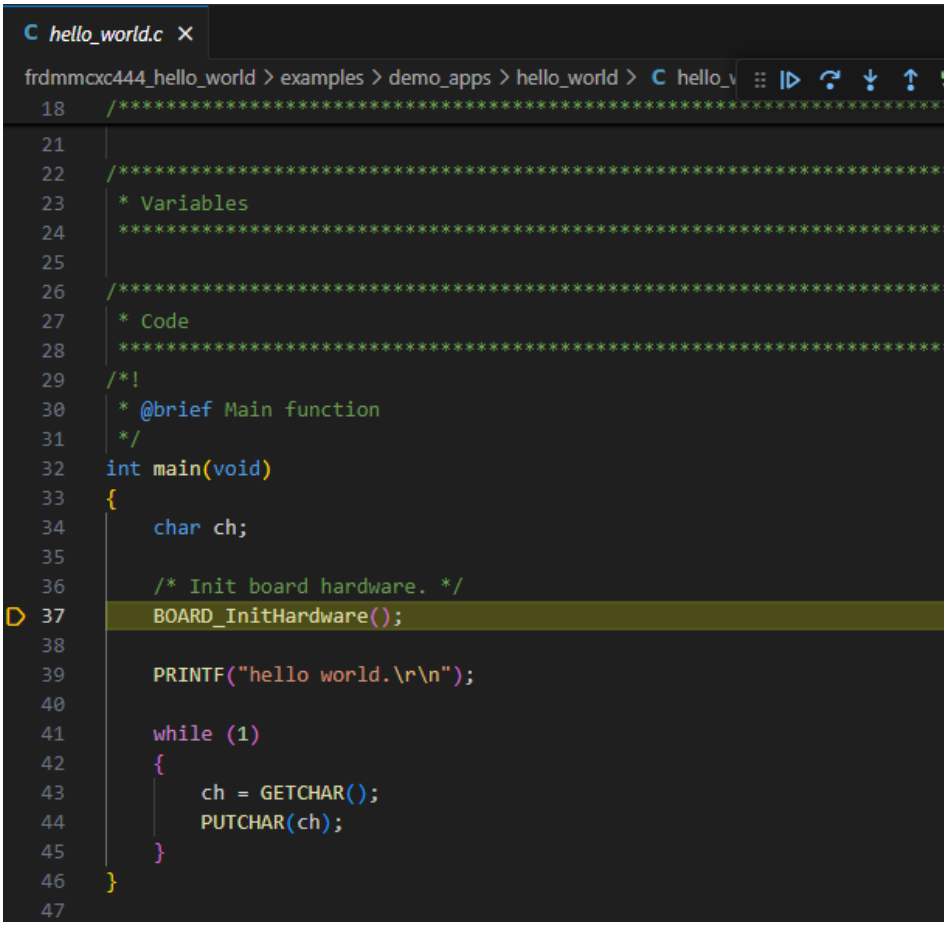


The debug session will begin with debug controls initially at the top of the interface.

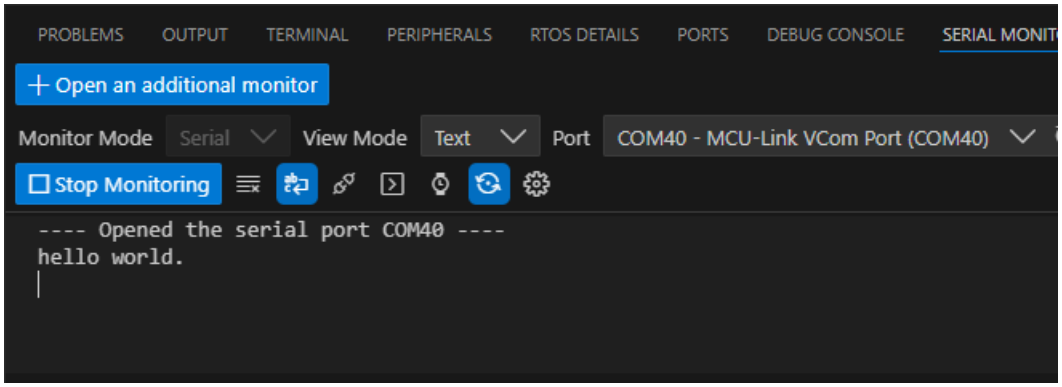
Debug Controls Use the debug controls to manage execution:

- **Continue**: Resume code execution
- **Step controls**: Navigate through code

- **Stop**: Terminate debug session



Monitor Output Observe application output in the **Serial Monitor** to verify correct operation.



Debug Probe Support For comprehensive information on debug probe support and configuration, refer to the MCUXpresso for VS Code Wiki DebugK section.

Project Configuration

Workspace Management The extension integrates with the MCUXpresso SDK workspace structure, providing access to:

- Example applications
- Board configurations

- Middleware components
- Build system integration

Multi-Project Support The PROJECTS view allows management of multiple imported projects within the same workspace.

Troubleshooting

Import Issues **SDK not detected:**

- Verify SDK workspace is properly initialized
- Ensure all required repositories are updated
- Check SDK manifest files are present

Project import failures:

- Confirm board support exists for selected example
- Verify toolchain installation
- Check example compatibility with selected board

Build Problems **Build failures:**

- Check integrated terminal for error messages
- Verify all dependencies are installed
- Ensure toolchain is properly configured

Debug Issues **Debug session fails:**

- Verify board connection via USB
- Check debug probe drivers are installed
- Confirm build completed successfully

Serial monitor problems:

- Verify correct VCom port selection
- Check baud rate configuration (115200)
- Ensure board drivers are installed

Integration with Command Line MCUXpresso for VS Code integrates with the underlying west build system, allowing seamless integration with command line workflows described in *Command Line Development*.

Advanced Features

Project Types The extension supports both repository-based and freestanding project types, providing flexibility in project organization and SDK integration.

Build System Integration The extension leverages the MCUXpresso SDK build system, providing access to all build configurations and options available through command line tools.

Next Steps

- Explore additional examples in the SDK
- Review *Command Line Development* for advanced build options
- Refer MCUXpresso for VS Code Wiki for detailed documentation
- Learn about *SDK Architecture* for better understanding of the development environment

Command Line Development This guide covers developing with the MCUXpresso SDK using command line tools and the west build system. This workflow applies to both GitHub Repository SDK and Repository-Layout SDK Package distributions.

Prerequisites

- GitHub Repository SDK workspace initialized OR Repository-Layout SDK Package extracted
- Development tools installed per *Installation Guide*
- Target board connected via USB

Understanding Board Support Use the west extension to discover available examples for your board:

```
west list_project -p examples/demo_apps/hello_world
```

This shows all supported build configurations. You can filter by toolchain:

```
west list_project -p examples/demo_apps/hello_world -t armgcc
```

Basic Build Commands

Standard Build Process Build with default settings (armgcc toolchain, first debug config):

```
west build -b your_board examples/demo_apps/hello_world
```

Specifying Build Configuration

```
# Release build
west build -b your_board examples/demo_apps/hello_world --config release

# Debug build with specific toolchain
west build -b your_board examples/demo_apps/hello_world --toolchain iar --config debug
```

Multicore Applications For multicore devices, specify the core ID:

```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config␣
↪flexspi_nor_debug
```

For multicore projects using sysbuild:

```
west build -b evkbmimxrt1170 --sysbuild ./examples/multicore_examples/hello_world/primary -Dcore_
↪id=cm7 --config flexspi_nor_debug --toolchain=armgcc -p always
```

Shield Support For boards with shields:

```
west build -b mimxrt700evk --shield a8974 examples/issdk_examples/sensors/fxls8974cf/fxls8974cf_poll -
↪Dcore_id=cm33_core0
```

Advanced Build Options

Clean Builds Force a complete rebuild:

```
west build -b your_board examples/demo_apps/hello_world -p always
```

Dry Run See what commands would be executed:

```
west build -b your_board examples/demo_apps/hello_world --dry-run
```

Device Variants For boards supporting multiple device variants:

```
west build -b your_board examples/demo_apps/hello_world --device MK22F12810 --config release
```

Project Configuration

CMake Configuration Only Run configuration without building:

```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world -Dcore_id=cm7 --cmake-only -p
```

Interactive Configuration Launch the configuration GUI:

```
west build -t guiconfig
```

Flashing and Debugging

Flash Application Flash the built application to your board:

```
west flash -r linkserver
```

Debug Session Start a debugging session:

```
west debug -r linkserver
```

IDE Project Generation Generate IDE project files for traditional IDEs:

```
# Generate IAR project
west build -b evkbmimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config␣
↪flexspi_nor_debug -p always -t guiproject
```

IDE project files are generated in mcuxsdk/build/<toolchain> folder.

Note: Ruby installation is required for IDE project generation. See *Installation Guide* for setup instructions.

Troubleshooting

Build Failures Use pristine builds to resolve dependency issues:

```
west build -b your_board examples/demo_apps/hello_world -p always
```

Toolchain Issues Verify environment variables are set correctly:

```
# Check ARM GCC
echo $ARMGCC_DIR
arm-none-eabi-gcc --version

# Check IAR (if using)
echo $IAR_DIR
```

Getting Help Display help information:

```
west build -h
west flash -h
west debug -h
```

Check Supported Configurations If unsure about supported options for an example:

```
west list_project -p examples/demo_apps/hello_world
```

Best Practices

Project Organization

- Keep custom projects outside the SDK tree
- Use version control for your application code
- Document any SDK modifications

Build Efficiency

- Use -p always for clean builds when troubleshooting
- Leverage --dry-run to understand build processes
- Use specific configs and toolchains to reduce build time

Development Workflow

1. Start with existing examples closest to your requirements
2. Copy and modify rather than building from scratch
3. Test with hello_world before moving to complex examples
4. Use configuration tools for pin muxing and clock setup

Next Steps

- Explore *VS Code Development* for integrated development experience
- Review *Workspace Structure* to understand SDK organization
- Refer build system documentation for advanced configurations

Using MCUXpresso Config Tools MCUXpresso Config tools provide a user-friendly way to configure hardware initialization of your projects. This guide explains the basic workflow with the MCUXpresso SDK west build system and the Config Tools.

Prerequisites

- GitHub Repository SDK workspace initialized OR Repository-Layout SDK Package extracted
- MCUXpresso Config Tools standalone installed (version 25.09 or above)
- MCUXpresso SDK Project that can be successfully built

Board Files MCUXpresso Config Tools generate source files for the board. These files include pin_mux.c/h and clock_config.c/h. The files contain initialization code functions that reflect the hardware configuration in the Config Tools. Within the SDK codebase, these files are specific for the board and either shared by multiple example projects or specific for one example. Open or import the configuration from the SDK project in the Config Tools and customize the settings to match the custom board or specific project use case and regenerate the code. See *User Guide for MCUXpresso Config Tools (Desktop)* (document GSMCUXCTUG) for details.

Note: When opening the configuration for SDK example projects, the board files may be shared across multiple examples. To ensure a separate copy of the board configuration files exists, create a freestanding project with copied board files.

Visual Studio Code To open the configuration in Visual Studio Code, use the context menu for the project to access Config Tools. See MCUXpresso Extension Documentation for details. Otherwise, use the manual workflow described in detail in the following section.

Manual Workflow Use the following steps:

1. Before using Config Tools, run the west command to get the project information for Config Tools from the SDK project files, for example:

```
west cfg_project_info -b lpcxpresso55s69 ...mcuxsdk/examples/demo_apps/hello_world/ -Dcore_
↪id=cm33_core0
```

   This results in the creation of the project information json file that is searched by the config tools when the configuration is created. The parameters of the command should match the build parameters that will be used for the project.

2. Launch the MCUXpresso Config Tools and in the **Start development** wizard, select **Create a new configuration based on the existing IDE/Toolchain project**. Select the created "cfg_tools" subfolder as a project folder (for example: ...mcuxsdk/examples/demo_apps/hello_world/cfg_tools/).

Updating the SDK West project **Note:** Updating project is supported with Config Tools V25.12 or newer only.

Changes in the Config tools generated source code modules may require adjustments to the toolchain project to ensure a successful build. These changes may mean, for example, adding the newly generated files, adding include paths, required drivers, or other SDK components. This section describes how to manually resolve the changes needed in the project within the toolchain projects based on the SDK project managed by the West tool.

After the configuration in the Config Tools is finished, write updated files to the disk using the 'Update Code' command. The written files include a json file with the required changes for the toolchain project.

To resolve the changes in the project in the terminal, launch the west command that updates the project. For example:

```
west cfg_resolve -b lpcxpresso55s69 ...mcuxsdk/examples/demo_apps/hello_world/ -Dcore_id=cm33_core0
```

This command updates the appropriate cmake and kconfig files to address the changes. After this, the application can be built.

Note: The cfg_resolve command supports additional arguments. Launch the *west cfg_resolve -h* command to get the list and description.

1.4 Release Notes

1.4.1 MCUXpresso SDK Release Notes

Overview

The MCUXpresso SDK is a comprehensive software enablement package designed to simplify and accelerate application development with Arm Cortex-M-based devices from NXP, including its general purpose, crossover and Bluetooth-enabled MCUs. MCUXpresso SW and Tools for DSC further extends the SDK support to current 32-bit Digital Signal Controllers. The MCUXpresso SDK includes production-grade software with integrated RTOS (optional), integrated enabling software technologies (stacks and middleware), reference software, and more.

In addition to working seamlessly with the MCUXpresso IDE, the MCUXpresso SDK also supports and provides example projects for various toolchains. The Development tools chapter in the associated Release Notes provides details about toolchain support for your board. Support for the MCUXpresso Config Tools allows easy cloning of existing SDK examples and demos, allowing users to leverage the existing software examples provided by the SDK for their own projects.

Underscoring our commitment to high quality, the MCUXpresso SDK is MISRA compliant and checked with Coverity static analysis tools. For details on MCUXpresso SDK, see MCUXpresso-SDK: Software Development Kit for MCUXpresso.

MCUXpresso SDK

As part of the MCUXpresso software and tools, MCUXpresso SDK is the evolution of Kinetis SDK, includes support for LPC, DSC,PN76, and i.MX System-on-Chip (SoC). The same drivers, APIs, and

middleware are still available with support for Kinetis, LPC, DSC, and i.MX silicon. The MCUXpresso SDK adds support for the MCUXpresso IDE, an Eclipse-based toolchain that works with all MCUXpresso SDKs. Easily import your SDK into the new toolchain to access to all of the available components, examples, and demos for your target silicon. In addition to the MCUXpresso IDE, support for the MCUXpresso Config Tools allows easy cloning of existing SDK examples and demos, allowing users to leverage the existing software examples provided by the SDK for their own projects.

In order to maintain compatibility with legacy Freescale code, the filenames and source code in MCUXpresso SDK containing the legacy Freescale prefix FSL has been left as is. The FSL prefix has been redefined as the NXP Foundation Software Library.

Development tools

The MCUXpresso SDK was tested with following development tools. Same versions or above are recommended.

SDK 25.12.00 is the final release that includes MCUXpresso IDE support for KW4x devices. Beginning with SDK 26.x.x, MCUXpresso IDE will no longer be supported for KW4x development, and MCUXpresso for Visual Studio Code will become the recommended integrated development environment.

- IAR Embedded Workbench for Arm, version is 9.60.4
- MCUXpresso IDE, Rev. 25.06.xx
- MCUXpresso for VS Code v25.09
- GCC Arm Embedded Toolchain 14.2.x

Supported development systems

This release supports board and devices listed in following table. The board and devices in bold were tested in this release.

Development boards	MCU devices
KW45EVK	KW45B41Z82AFPA, KW45B41Z82AFTA, KW45B41Z83AFPA, **KW45B41Z83AFTA**, KW45B41Z52AFPA, KW45B41Z52AFTA, KW45B41Z53AFPA, KW45B41Z53AFTA, KW45Z41052AFPA, KW45Z41052AFTA, KW45Z41053AFPA, KW45Z41053AFTA, KW45Z41082AFPA, KW45Z41082AFTA, KW45Z41083AFPA, KW45Z41083AFTA

MCUXpresso SDK release package

The MCUXpresso SDK release package content is aligned with the silicon subfamily it supports. This includes the boards, CMSIS, devices, middleware, and RTOS support.

Device support The device folder contains the whole software enablement available for the specific System-on-Chip (SoC) subfamily. This folder includes clock-specific implementation, device register header files, device register feature header files, and the system configuration source files. Included with the standard SoC support are folders containing peripheral drivers,

toolchain support, and a standard debug console. The device-specific header files provide a direct access to the microcontroller peripheral registers. The device header file provides an overall SoC memory mapped register definition. The folder also includes the feature header file for each peripheral on the microcontroller. The toolchain folder contains the startup code and linker files for each supported toolchain. The startup code efficiently transfers the code execution to the main() function.

Board support The boards folder provides the board-specific demo applications, driver examples, and middleware examples.

Demo application and other examples The demo applications demonstrate the usage of the peripheral drivers to achieve a system level solution. Each demo application contains a readme file that describes the operation of the demo and required setup steps. The driver examples demonstrate the capabilities of the peripheral drivers. Each example implements a common use case to help demonstrate the driver functionality.

RTOS

FreeRTOS Real-time operating system for microcontrollers from Amazon

Middleware

GenFSK link layer The Generic FSK protocol enables radio operation using a custom GFSK/GMSK or MSK modulation format.

Main Features supported:

- Highly configurable packet structure
- Optimized Sequence Command Set
- High-precision timebase to maintain network timing
- Two timer-compare mechanisms for Interrupt Generation and Sequence Launching
- Hardware automation for packet transmit and receive, CRC and Whitening
- Up to four network addresses to synchronize to, can be 8-bit, 16-bit or 32-bit
- Packet Lengths up to 2047 Bytes
- Support complex auto-sequence, like CCA before TX, Auto-ACK, TR.
- Many operating modes can support the sending and receiving of multiple protocol packets, such as Bluetooth LE.

Wireless XCVR The XCVR component provides a base Transceiver Driver for the 2.4 GHz narrowband radio.

Bluetooth LE Controller

- Main features supported:
  - Peripheral Role
  - Central Role
  - Multiple PHYs (1 Mbps, 2 Mbps, Coded PHY)
  - Asymmetric Connections

  - Public/Random/Static Addresses
  - Network/Device Privacy Modes
  - Extended Advertising
  - Extended Scanning
  - Passive/Active Scanning
  - LE Encryption
  - LE Ping Procedure
  - HCI Test Interface
  - UART Test Interface
  - Randomized Advertising Channel Indexing
  - Sleep Clock Accuracy Update - Mechanism
  - ADI Field in Scan Response Data
  - HCI Support for Debug Keys in LE - Secure Connections
- Main capabilities supported:
  - Simultaneous scanning 1 Mbps and Long Range
  - Scanning and advertising in parallel
  - 24 connections as a central role
  - 24 connections as a peripheral role
  - Any combination of central and peripheral roles (24 connections maximum)
  - 8 connections with a 7.5 ms connection interval
  - Two advertising sets in parallel (\Four adv set as Early Access Release).
  - 26 Accept List entries
  - 36 Resolvable Private Address (RPA) entries
  - Up to two Chain Packets per Extended Advertising set
  - Enhanced Notification on end of - Scanning/Advertising/Connection events
  - Connection event counters associated to Bluetooth LE packet reception
  - Timestamp associated to Bluetooth LE packet reception
  - RF channel info associated to Bluetooth LE packet reception
  - NXP proprietary Bluetooth LE Handover feature
  - Decision Based Advertising Filtering (DBAF)
  - Advertising Coding Selection (ACS)
  - Periodic Advertising with Responses (PAwR) Additional features supported for KW47 and MCX W72 devices:
  - Channel Sounding Additional features supported as EAR (\Early Access Release) in the KW47 experimental build:
  - Channel Sounding TX/SNR. Additional features supported as EAR (\Early Access Release) in the KW45/KW47 experimental builds:
  - LE Test Mode Enhancement (\UTP/OTA).
  - LL Extended Feature Set.
  - Monitoring Advertisers.

– Randomized Resolvable Private Address (\RPA).

Note: Project configuration enabling Experimental features on KW45 and MCX W71 requires the Radio Subsystem (NBU) Firmware to be reprogrammed with the firmware provided in the SDK under *middleware\wireless\ble_controller\bin\experimental*. For NBU programming steps, see the *EVK Quick Start Guide* and Secure Provisioning SDK (SPSDK) documentation.

Project configurations that require usage of the Bluetooth LE controller including all Bluetooth LE examples require the Radio Subsystem (NBU) Firmware to be re-programmed with the firmware provided in the SDK under middleware\wireless\ble_controller\bin.

Connectivity framework The Connectivity Framework is a software component that provides hardware abstraction modules to the upper layer connectivity stacks and components. It also provides a list of services and APIs (see *Supported services*). The Connectivity Framework modules are located in the *middleware\wireless\framework SDK* folder.

Supported services

- **FSCI** - Framework Serial Communication Interface
- **FunctionLib** - Common function library utilities
- **HWParameter** - Hardware parameter management
- **LowPower** - Low power mode management
- **ModuleInfo** - Module information and versioning
- **NVS** - Non-Volatile Storage
- **NVM** - Non-Volatile Memory management
- **OtaSupport** - Over-The-Air update support
- **SecLib_RNG** - Security library and Random Number Generator
- **Sensors** - Sensor abstraction layer
- **SFC** - Smart Frequency Calibration
- **WorkQ** - Work queue management

Supported platform

- **KW45_MCXW71**
- **KW47_MCXW72**
- **MCXW23**
- **RW61X**
- **RT1060** and **RT1170**
- **i.MX RT595s**

Advanced features supported on platforms

KW45_MCXW71

- FRO32K with smart frequency calibration (see SFC)
- Power down mode support (for evaluation only)

KW47_MCXW72

- FRO32K with smart frequency calibration (see SFC)
- Power down mode support (for evaluation only)
- Crystal 32M trimming with temperature
- Debug module for NBU
- Extended NBU support with SecLib and pseudo RNG support

CMSIS DSP Library The MCUXpresso SDK is shipped with the standard CMSIS development pack, including the prebuilt libraries.

TF-M Trusted Firmware - M Library

PSA Test Suite Arm Platform Security Architecture Test Suite

secure_storage secure_storage

EdgeLock SE050 Plug and Trust Middleware Secure subsystem library - SSS APIs

Multicore Multicore Software Development Kit

NXP IoT Agent NXP IoT Agent

mbedTLS mbedtls SSL/TLS library v3.x

mbedTLS mbedtls SSL/TLS library v2.x

LittleFS LittleFS filesystem stack

LIN Stack LIN Stack middleware

FreeMASTER FreeMASTER communication driver for 32-bit platforms.

NXP PSA CRYPTO DRIVER PSA crypto driver for crypto library integration via driver wrappers

Release contents

Provides an overview of the MCUXpresso SDK release package contents and locations.

Deliverable	Location
Boards	INSTALL_DIR/boards
Demo Applications	INSTALL_DIR/boards/<board_name>/demo_apps
Driver Examples	INSTALL_DIR/boards/<board_name>/driver_examples
eIQ examples	INSTALL_DIR/boards/<board_name>/eiq_examples
Board Project Template for MCUXpresso IDE NPW	INSTALL_DIR/boards/<board_name>/project_template
Driver, SoC header files, extension header files and feature header files, utilities	INSTALL_DIR/devices/<device_name>
CMSIS drivers	INSTALL_DIR/devices/<device_name>/cmsis_drivers
Peripheral drivers	INSTALL_DIR/devices/<device_name>/drivers
Toolchain linker files and startup code	INSTALL_DIR/devices/<device_name>/<toolchain_nam
Utilities such as debug console	INSTALL_DIR/devices/<device_name>/utilities
Device Project Template for MCUXpresso IDE NPW	INSTALL_DIR/devices/<device_name>/project_templat
CMSIS Arm Cortex-M header files, DSP library source	INSTALL_DIR/CMSIS
Components and board device drivers	INSTALL_DIR/components
RTOS	INSTALL_DIR/rtos
Release Notes, Getting Started Document and other documents	INSTALL_DIR/docs
Tools such as shared cmake files	INSTALL_DIR/tools
Middleware	INSTALL_DIR/middleware

What is new

The following changes have been implemented compared to the previous SDK release version (25.12.00-pvw2).

- **Bluetooth LE Host Stack and Applications**

  #### Added

  - ‘Monitoring Advertisers’ support in the ‘fsci_black_box’ and ‘ble_shell’ applications

  #### Fixed

  - Missing handler for Version2 of the Set RPA Timeout command
  - Details can be found in github repository **nxp-mcuxpresso/mcuxsdk-middleware-bluetooth-host/CHANGELOG.md**.

- **Bluetooth LE controller**
  - Periodic Advertising updates:
    - Fixed PerAdvSync skip scheduling.
    - Improved PAwR Responses (TX & RX) instant.
    - Fixed calculation of the count parameters in Periodic Advertising Subevent Data Request.
  - Fixed a corner case where LL_CONNECTION_PARAM_REQ is immediately followed by LL_TERMINATE_IND.

- **Transceiver Drivers (XCVR)**

  - Added API to control PA ramp type and duration.

- **Connectivity framework**

  Major Changes - [wireless_mcu][wireless_nbu] Replaced interrupt masking macros with static inline functions PLATFORM_SetInterruptMask() and PLATFORM_ClearInterruptMask() to ensure consistent BASEPRI value handling across all compilers. This addresses compiler-dependent behavior issues with the previous macro implementation. - [wireless_mcu] Added BASEPRI-based interrupt masking in PLATFORM_RemoteActiveRel() to allow high-priority IRQs while ensuring only IMU0 or thread context can call this function. - [wireless_mcu] Introduced gPlatformUseHwParameter_d compile flag to allow builds without HWParameter section. When undefined or set to 0, crystal trimming functions conditionally access HWParameters only when required. **Minor Changes** - [wireless_mcu] Set RL_BUFFER_PAYLOAD_SIZE to word-aligned value as expected by rpmsg-lite. - [wireless_mcu] Added system-generated HCI vendor events capability for debug and diagnostic purposes. - [Platform] Simplified enablement of reset features via pin detection - Automatically selects gUseResetByLvdForce_c when gAppForceLvdResetOnResetPinDet_d is enabled. - Automatically select gUseResetByDeepPowerDown_c when gAppForceDeepPowerDownResetOnResetPinDet_d is enabled. - [RNG] Replaced gRngHasSecLibDependency_d compilation switch with gRngUseSecLib_d. **Bug fixes** - [wireless_mcu] Fixed race condition in PLATFORM_RemoteActiveRel() by adding verification loop to confirm NBU core execution before releasing power domain. - [wireless_mcu] Added instruction synchronization barrier (__ISB()) after interrupt re-enable in PLATFORM_RemoteActiveRel() to ensure pending interrupts execute between critical sections. - [wireless_mcu] Fixed external IO voltage isolation issue during low-power initialization - isolation is now cleared at init to ensure proper behavior. - [wireless_mcu] Replaced spin-wait loops with event-based synchronization in NBU communication APIs. Added mutex protection to PLATFORM_NbuApiReq() and PLATFORM_GetNbuInfo() to prevent race conditions and deadlocks when multiple tasks call these APIs concurrently. - [wireless_mcu] Fixed OSA bare metal event race condition in ICS where auto-clear event feature could cause tasks to become permanently stuck. Disabled auto-clear feature in bare metal builds and manually clear event flags after OSA_EventWait() returns successfully. - [NVM] Fixed NvIdle() to prevent looping for more operations than the queue size. - [NVS] Fixed blank check procedure to return false (non-blank) when checking a 0 length area. - [NVS] Made external and internal flash ports consistent. - [MISRA] Various MISRA compliance fixes in NVM, HWParameter, LowPower, SecLib, Platform modules and IFR offset definitions.

Known issues

This section lists the known issues, limitations, and/or workarounds.

FRO6M Clock Stability Issue

According to ERRATA ERR052742, the FRO6M clock is not stable on some parts. FRO6M outputs lower frequency signal instead of 6MHz when device is reset or wakes up from low power. It can impact peripherals using it as a clock source.

Impact on TSTMR Module The TSTMR (Time Stamp Timer) module exclusively uses the FRO6M clock source. Due to the aforementioned stability issues, **avoid using TSTMR-related APIs if your application requires high-precision timing**.

Recommendation For applications requiring precise timing, consider using alternative timer modules that support more stable clock sources.

New project wizard compile failure

The following components request the user to manually select other components that they depend upon in order to compile. These components depend on several other components and the New Project Wizard (NPW) is not able to decide which one is needed by the user.

Note: xxx means core variants, such as, cm0plus, cm33, cm4, cm33_nodsp.

Also for low-level adapter components, currently the different types of the same adapter cannot be selected at the same time. For example, if there are two types of timer adapters, gpt_adapter and pit_adapter, only one can be selected as timer adapter in one project at a time. Duplicate implementation of the function results in an error.

Only FreeRTOS is tested for RTOS support

This release only supports the FreeRTOS kernel and a bare-metal non-preemptive task scheduler.

Bluetooth LE

Most sensor applications have pairing and bonding disabled to allow a faster interaction with mobile applications. These two security features can be enabled in the app_preinclude.h header file.

Bluetooth LE controller:

- The maximum Advertising data length is limited to 800 bytes.
- The scanner may sporadically miss some chained packets.

Periodic Advertising with Responses (PAwR):

- Periodic Advertising with Response (PAwR) is not supported with the configuration "Subevent Interval = Number of Response Slots x Response Slot Spacing with Response Slot Spacing = 0x2".
- Periodic Advertising placement with connection events is unoptimized.
- The feature is not functional with the Free-Running Oscillator (FRO32K); it requires a 32 KHz Crystal Oscillator with accuracy less than 50 ppm.

KW45/MCXW71: No specific issues.

KW47/MCXW72: Channel Sounding (CS): Limitations:

- RTT with Sounding Sequence is not supported.
- LE 2M 2BT PHY is not supported.
- Maximum 6 Channel Sounding procedures are supported in parallel.
- Scheduling of activities may be non optimal when multiple Channel Sounding procedures are running in parallel.
- Phase measurement bias is within certification range (<1.7x2πns) with KW47 EVK board. However, if different PCB or antenna matching is used, some bias may appear due to increased delay. Known issues:
- When CS Subevents are configured very close from each other (<700us), some Subevents may be aborted with reason 0x3.
- When CS offset is configured too close from ACL anchor point, the anchor point may not be served (TX on central or RX on peripheral will not happen). Ideally, CS Offset should be configured greater than 1ms.

- RTT bias compensation:
  - For parts not properly configured at production (IFR blank), RTT bias may not be compensated properly. Consequently, an inaccuracy of +/-2m may be observed.

LIN New Project Wizard (NPW) issue

- The lin (LIN Driver) and lin_stack (LIN Stack Driver) drivers components should not be enabled at the same time while creating the new projects in MCUXpresso. Otherwise there will be the compiling issue.
- The lin_stack (LIN Stack Driver) is not actually a driver. It is an adapt layer for LIN Stack middleware to adapt to the low level lpuart driver and cannot be used in NPW alone. So, select the LIN Stack middleware and then the lin_stack is selected automatically since it is required by LIN Stack middleware. Besides, customer need to add the lin_cfg.c/h in application layer for user definition of frame data and add **FSL_SDK_LIN_STACK_ENABLE=1** in MCUXpresso preprocessor, otherwise the compiling of LIN Stack will report error.

Flash ROMAPI

Note that:

- If using ROM API for internal flash or SPI NOR operation, reserve RAM location 0x200030A0 - 0x200032CF (0x300030A0 - 0x300032CF).
- If using kb API, reserve 0x20002000 - 0x200032FF (0x30002000 - 0x300032FF).

Other limitations

- The following Connectivity Framework configurations are Experimental and not recommended for mass production:
  - Power down on application power domain.
- GenFSK Connectivity_test application is not operational with Low Power enabled.
- Serial manager is only supported on UART (not I2C nor SPI).
- The --no-warn-rwx-segments cannot been recognized on legacy MCUXpresso IDE versions.

  The --no-warn-rwx-segments option in MCUXpresso projects should be manually removed from the project settings if someone needs to use legacy (< 11.8.0) MCUXpresso IDE versions
- If the FRO32K is configured as the clock source of the CM33 Core then the debug session will block in both IAR, MCUX CMSIS-DAP while debugging. Use a lower debug wire speed, for example 1 MHz instead of the default one.

  In IAR, the option is in **Runtime Checking** -> **Debugger** -> **CMSIS DAP** -> **Interface** -> **Interface speed**.

  In MCUXpresso IDE, the option is in **LinkServer Debugger** -> **Advanced Settings** -> **Wire-speed (Hz)**.

Examples hello_world_ns, secure_faults_ns, and secure_faults_trdc_ns have incorrect library path in GUI projects

When the affected examples are generated as GUI projects, the library linking the secure and non-secure worlds has an incorrect path set. This causes linking errors during project compilation.

Examples: hello_world_ns, hello_world_s, secure_faults_ns, secure_faults_s, secure_faults_trdc_ns, secure_faults_trdc_s

Affected toolchains: mdk, iar

Workaround: In the IDE project settings for the non-secure (_ns) project, find the linked library (named hello_world_s_CMSE_lib.o, or similar, depending on the example project) and replace the path to the library with <build_directory>/<secure_world_project_folder>/<IDE>/, replacing the subdirectory names with the build directory, the secure world project name, and IDE name.

El2go examples on kw45b41zevk

The el2go examples are executed on socketed board with KW45 samples instead of actual EVKs.

Examples: el2go_blob_test, el2go_import_blob

Affected toolchains: All

1.5 ChangeLog

1.5.1 MCUXpresso SDK Changelog

Board Support Files

board

[25.06.00]

- Initial version

clock_config

[25.06.00]

- Initial version

pin_mux

[25.06.00]

- Initial version

CACHE LPCAC

[2.1.2]

- Improvements
  - Add memory barrier when enabling/disabling cache.

[2.1.1]

- Bug Fixes
  - MISRA C-2012 issue fixed: rule 10.1.

[2.1.0]

- Bug Fixes
  - Deleted L1CACHE_EnableCodeCacheWriteBuffer function because of no enable bit in register CPCR2.

[2.0.0]

- Initial version.

CCM32K

[2.2.1]

- Bug Fixes
  - Fixed doxygen warning: unbalanced grouping commands.

[2.2.0]

- Improvements
  - Removed wait loops in CCM32K's APIs.

[2.1.1]

- Improvements
  - Updated the judgment conditions in the CCM32K_Enable32kFro() API to avoid code stuck in the while loop.

[2.1.0]

- New Features
  - Added clock monitor control function group and clock gate control function group to support some devices that support clock monitor and clock gate control.

[2.0.0]

- Initial version.

CLOCK

[2.2.5]

- Bug Fixes:
  - Fixed violations of the CERT INT31-C.

[2.2.4]

- Improvements:
  - Added comments for ERR052742: FRO6M clock is not stable.

[2.2.3]

- Improvements:
  - Fixed the MSG issue that overrunning array fro192mFreq of 5 4-byte elements at element index 7.

[2.2.2]

- New Features:
  - Provide the new API CLOCK_IsFIRCAutoTrimLocked to check whether FIRC auto trim locked to target frequency range.

[2.2.1]

- Bug fix:
  - Fixed the GPIO_PinInit function may cause the hardfault in clock driver when pass the GPIOD as the fist parameter.

[2.2.0]

- Improvements
  - Added the support for TPM2.

[2.1.0]

- Improvements
  - Provide the FSL_SDK_FORCE_CLK_DRIVER_NS_ACCESS macro to force use the none secure address even the trustzone secure mode is enabled.

[2.0.0]

- Fixed the MISRA issues.

[1.1.0]

- Some minor fixes.

[1.0.0]

- Initial version.

CMC

[2.4.4]

- Bug Fixes
  - Fixed doxygen warning: unbalanced grouping commands.

[2.4.3]

- Improvements
  - Add timeout for while loop code.

[2.4.2]

- Improvements
  - Add support for specific KW4x device variants
    - Let kCMC_SecurityViolationReset depend on whether CMC_SRS_SECVIO_MASK definition exists or not.
    - CMC_LockWriteOperationToBootRomStatusReg and CMC_CheckBootRomStatusRegWriteLocked exist only if FSL_FEATURE_CMC_HAS_NO_BOOTROM_LOCK_REGISTER is not defined

[2.4.1]

- Improvements
  - Clear SCB SCR[SLEEPDEEP] bitfield after wakeup.

[2.4.0]

- New Features
  - Added new interface to support CMC[DIER] register.
- Improvements
  - Updated _cmc_system_sram_arrays enumeration to support some devices that provide more sram bank.

[2.3.1]

- Bug Fixes
  - Fixed violations of MISRA C-2012 rule 10.7;

[2.3.0]

- Improvements
  - Added new functions to support controls of BLR register.
  - Updated CMC_GetBootRomStatus() to support some devices that equipped multiple BootROM status registers.
  - Added CMC_WriteBootRomStatusReg() to write BootRom Status register.

[2.2.1]

- Improvements
  - For some devices, SRS_JTAG bit is reserved. Added a feature macro to adapt to different devices.

[2.2.0]

- Improvements
  - Updated _cmc_system_sram_arrays enumeration, make it more universal.
  - Updated SRAM related APIs(CMC_PowerOffSRAMAllMode(), CMC_PowerOffSRAMLowPowerOnly(), CMC_PowerOnSRAMAllMode(), CMC_PowerOnSRAMLowPowerOnly()), due to updates of registers' names.
  - Renamed CMC_GetBootConfigPinLogic() to CMC_GetISPMODEPinLogic().
  - Renamed CMC_ClearBootConfig() to CMC_ClearISPModePinLogic().
  - Updated enumeration _cmc_power_mode_protection due to some macros are deleted in header file.

[2.1.0]

- Improvements
  - Added some macros to separate the scenes that some registers are reserved in some devices.

[2.0.0]

- Initial version.

COMMON

[2.6.3]

- Bug Fixes
  - Fixed build issue of CMSIS PACK BSP example caused by CMSIS 6.1 issue.

[2.6.2]

- Bug Fixes
  - Fixed violations of MISRA C-2012 rule for implicit conversions in boolean contexts

[2.6.1]

- Improvements
  - Support Cortex M23.

[2.6.0]

- Bug Fixes
  - Fix CERT-C violations.

[2.5.0]

- New Features
  - Added new APIs InitCriticalSectionMeasurementContext, DisableGlobalIRQEx and EnableGlobalIRQEx so that user can measure the execution time of the protected sections.

[2.4.3]

- Improvements
  - Enable irqs that mount under irqsteer interrupt extender.

[2.4.2]

- Improvements
  - Add the macros to convert peripheral address to secure address or non-secure address.

[2.4.1]

- Improvements
  - Improve for the macro redefinition error when integrated with zephyr.

[2.4.0]

- New Features
  - Added EnableIRQWithPriority, IRQ_SetPriority, and IRQ_ClearPendingIRQ for ARM.
  - Added MSDK_EnableCpuCycleCounter, MSDK_GetCpuCycleCount for ARM.

[2.3.3]

- New Features
  - Added NETC into status group.

[2.3.2]

- Improvements
  - Make driver aarch64 compatible

[2.3.1]

- Bug Fixes
  - Fixed MAKE_VERSION overflow on 16-bit platforms.

[2.3.0]

- Improvements
  - Split the driver to common part and CPU architecture related part.

[2.2.10]

- Bug Fixes
  - Fixed the ATOMIC macros build error in cpp files.

[2.2.9]

- Bug Fixes
  - Fixed MISRA C-2012 issue, 5.6, 5.8, 8.4, 8.5, 8.6, 10.1, 10.4, 17.7, 21.3.
  - Fixed SDK_Malloc issue that not allocate memory with required size.

[2.2.8]

- Improvements
  - Included stddef.h header file for MDK tool chain.
- New Features:
  - Added atomic modification macros.

[2.2.7]

- Other Change
  - Added MECC status group definition.

[2.2.6]

- Other Change
  - Added more status group definition.
- Bug Fixes
  - Undef __VECTOR_TABLE to avoid duplicate definition in cmsis_clang.h

[2.2.5]

- Bug Fixes
  - Fixed MISRA C-2012 rule-15.5.

[2.2.4]

- Bug Fixes
  - Fixed MISRA C-2012 rule-10.4.

[2.2.3]

- New Features
  - Provided better accuracy of SDK_DelayAtLeastUs with DWT, use macro SDK_DELAY_USE_DWT to enable this feature.
  - Modified the Cortex-M7 delay count divisor based on latest tests on RT series boards, this setting lets result be closer to actual delay time.

[2.2.2]

- New Features
  - Added include RTE_Components.h for CMSIS pack RTE.

[2.2.1]

- Bug Fixes
  - Fixed violation of MISRA C-2012 Rule 3.1, 10.1, 10.3, 10.4, 11.6, 11.9.

[2.2.0]

- New Features
  - Moved SDK_DelayAtLeastUs function from clock driver to common driver.

[2.1.4]

- New Features
  - Added OTFAD into status group.

[2.1.3]

- Bug Fixes
  - MISRA C-2012 issue fixed.
    - Fixed the rule: rule-10.3.

[2.1.2]

- Improvements
  - Add SUPPRESS_FALL_THROUGH_WARNING() macro for the usage of suppressing fallthrough warning.

[2.1.1]

- Bug Fixes
  - Deleted and optimized repeated macro.

[2.1.0]

- New Features
  - Added IRQ operation for XCC toolchain.
  - Added group IDs for newly supported drivers.

[2.0.2]

- Bug Fixes
  - MISRA C-2012 issue fixed.
    - Fixed the rule: rule-10.4.

[2.0.1]

- Improvements
  - Removed the implementation of LPC8XX Enable/DisableDeepSleepIRQ() function.
  - Added new feature macro switch "FSL_FEATURE_HAS_NO_NONCACHEABLE_SECTION" for specific SoCs which have no noncacheable sections, that helps avoid an unnecessary complex in link file and the startup file.
  - Updated the align(x) to **attribute**(aligned(x)) to support MDK v6 armclang compiler.

[2.0.0]

- Initial version.

CRC

[2.0.5]

- Bug fix:
  - Fix CERT-C issue with boolean-to-unsigned integer conversion.

[2.0.4]

- Improvements
  - Release peripheral from reset if necessary in init function.

[2.0.3]

- Bug fix:
  - Fix MISRA issues.

[2.0.2]

- Bug fix:
  - Fix MISRA issues.

[2.0.1]

- Bug fix:
  - DATA and DATALL macro definition moved from header file to source file.

[2.0.0]

- Initial version.

EDMA (DMA3)

[2.5.3]

- Bug Fixes
  - Fixed coverity issues with CERT INT30-C, CERT INT31-C compliance.
  - Fixed violations of MISRA C-2012 rule 10.4, 8.5.

[2.5.2]

- Bug Fixes
  - Fixed the EDMA header index retrieval error caused by done bit calculation mistake issue.

[2.5.1]

- Bug Fixes
  - enables the auto stop request feature in API EDMA_ResetChannel

[2.5.0]

- Improvements
  - Add API EDMA_GetTransferSize for EDMA_PrepareTransferConfig to reduce HIS CCM value.

[2.4.0]

- Improvements
  - Added peripheral to peripheral support in DMA3 driver.
- Bug Fixes
  - Fixed the ERQ bit reading error issue.

[2.3.2]

- Improvements
  - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.3.1]

- Bug Fixes
  - Added clear TCD_CITER_ELINKNO and TCD_BITER_ELINKNO registers in EDMA_AbortTransfer to make sure the TCD registers in a correct state for next calling of EDMA_SubmitTransfer.

[2.3.0]

- Improvements
  - Added feature FSL_FEATURE_EDMA_HAS_NO_SBR_ATTR_BIT to separate DMA without ATTR bitfield.

[2.2.7]

- Bug Fixes
  - Fixed violations of MISRA C-2012 rule 10.8, 5.6.

[2.2.6]

- Bug Fixes
  - Fixed the TCD overwrite issue when submit transfer request in the callback if there is a active TCD in hardware.

[2.2.5]

- Bug Fixes
  - Fixed violations of MISRA C-2012 rule 10.1, 10.4.

[2.2.4]

- Bug Fixes
  - Fix the issue that EDMA_AbortTransfer not reset edma handle(tcdUsed, tail, header) and fix EDMA_InstallTCDMemory(handle->header = 1)

[2.2.3]

- Improvements
  - Added feature FSL_FEATURE_EDMA_MODULE_CHANNEL_IRQ_ENTRY_SUPPORT_PARAMETER to support driver IRQ handler with parameters.
  - Added feature FSL_FEATURE_EDMA_HAS_COMMON_CLOCK_GATE to improve clock gate control in dma driver.

[2.2.2]

- Bug Fixes
  - Fixed the issue of EDMA_SubmitTransfer return busy when calling EDMA_EnableChannelInterrupts before submit transfer.
  - Fixed violations of MISRA C-2012 rule 10.4, 10.1, 9.2, 10.4, 10.6, 14.4, 10.7, 14.3, 11.6.

[2.2.1]

- Improvements
  - Removed channel MUX reset from EDMA_ResetChannel, since channel mux should be constant while channel is alive.

[2.2.0]

- Improvements
  - Added new API EDMA_SetChannelMux to support channel mux feature.
  - Added new API EDMA_PrepareTransferConfig to expose paramters source offset and destination offset.
  - Exposed EDMA_InstallTCD function to application.

  - Added source/destination address alignment check.

[2.1.1]

- Improvements
  - Added 8bytes transfer width feature support in driver.

[2.1.0]

- Bug Fixes
  - Added const type for parameter configuration in EDMA_SubmitTransfer and EDMA_HandleTransferConfig API.
  - Added configurations for srcAddr and destAddr in EDMA_PrepareTransfer API.

[2.0.2]

- Improvements
  - Updated eDMA driver to support MP_CR bit GMRC.
  - Updated eDMA instance name for i.MX 8QM.
  - Used instance number as factor to calculate channel number for different instance instead of hard code.

[2.0.1]

- New Features
  - Added control macro to enable/disable the CLOCK code in current driver.
  - Added s_EDMAEnabledChannel to record enabled channel to merge all the channel IRQ handler into driver IRQ handler.
  - Added feature macro for bits EMI and EBW in MP_CSR.
- Improvements
  - Removed all the separated channel IRQ handler in DMA driver.

[2.0.0]

- Initial version.

ELEMU

- 2.1.3 Fix CERT INT30-C and CERT INT31-C findings.
- 2.1.2 Fix macro BIT redefined issue when compiling with Zephyr.
- 2.1.1 Fix MISRA issues.
- 2.1.0 Cleanup of unused legacy FW. Updated ELEMU_loadFw function.
- 2.0.0 Renamed MU to ELEMU to avoid confusion with other Messaging units.
- 0.0.4 Updated register name according to latest device header file.
- 0.0.3 Fix MISRA issues.

- 0.0.2 The name of SNT has changed to ELEMU.
- 0.0.1 Initial version of SNT driver.

EWM

[2.0.4]

- Bug Fixes
  - Fixed CERT INT31-C violations.

[2.0.3]

- Bug Fixes
  - Fixed violation of MISRA C-2012 rules: 10.1, 10.3.

[2.0.2]

- Bug Fixes
  - Fixed violation of MISRA C-2012 rules: 10.3, 10.4.

[2.0.1]

- Bug Fixes
  - Fixed the hard fault in EWM_Deinit.

[2.0.0]

- Initial version.

FLASH

[2.3.3]

- Improvements
  - Added timeout for while loop in flash_command_pre_sequence, flash_command_sequence, flash_erase_sequence, flash_pgm_sequence and flash_command_pre_sequence API.

[2.3.2]

- Improvement
- Enabled the kFLASH_PropertyPflash1SectorSize properity support in FLASH_GetProperty API.

[2.3.1]

- Improvement
- Clear flash cache before every erase to prevent the possibility of returning stale data.

[2.3.0]

- New features
- Add support for MCX N series

[2.2.1]

- Improvement
  - To fix the potential unsigned integer overflow in the operation, added a check to ensure that the addition does not exceed the maximum value.

[2.2.0]

- New Features
  - Add the support for MW30.

[2.1.2]

- Improvement
  - Add the support for KW47/MCXW72 phantoms.

[2.1.1]

- Improvement
  - Add the conditional compiling flag '#if defined(RF_FMU)' to make the driver be compatible with the non-radio phantoms.

[2.1.0]

- Bug Fixes
  - Fix flash driver run error in flash memory

[2.0.0]

- Initial version.

FLEXCAN

[2.14.5]

- Improvements
  - Make API `FLEXCAN_GetFDMailboxOffset` public.
  - Add API `FLEXCAN_SetMbID` and `FLEXCAN_SetFDMbID` to configure Message Buffer ID individually.
- Bug Fixes
  - Fixed violations of the CERT INT30-C INT31-C.
  - Fixed violations of the CERT ARR30-C.

[2.14.4]

- Bug Fixes
  - Fixed violations of the MISRA C-2012 rules 8.4, 10.1, 10.4, 18.1.

[2.14.3]

- Improvements
  - Add unhandled interrupt events check for following API:
    * FLEXCAN_MbHandleIRQ
    * FLEXCAN_EhancedRxFifoHandleIRQ
- Bug Fixes
  - Remove FLEXCAN_MemoryErrorHandleIRQ on some platform without memory error interrupt.
  - Add conditional compile for CTRL2[ISOCANFDEN] because some platform do not have this bit.

[2.14.2]

- Improvements
  - Add Coverage Justification for uncovered code.
  - Adjust API FLEXCAN_TransferAbortReceive order.
  - Update FLEXCAN_Enable to enter Freeze Mode first when enter Disable mode on some platform.
  - Added while loop timeout for following API:
    * FLEXCAN_EnterFreezeMode
    * FLEXCAN_ExitFreezeMode
    * FLEXCAN_Enable
    * FLEXCAN_Reset
    * FLEXCAN_TransferSendBlocking
    * FLEXCAN_TransferReceiveBlocking
    * FLEXCAN_TransferFDSendBlocking
    * FLEXCAN_TransferFDReceiveBlocking
    * FLEXCAN_TransferReceiveFifoBlocking
    * FLEXCAN_TransferReceiveEnhancedFifoBlocking
- Bug Fixes
  - Remove remote frame feature in CANFD mode because there is no remote frame in the CANFD format.
  - Remove legacy Rx FIFO disabled branch in FLEXCAN_SubHandlerForLegacyRxFIFO and FLEXCAN_SubHandlerForDataTransfered.

[2.14.1]

- Bug Fixes
  - Fixed register IMASK2-4 IFLAG2-4 HR_TIME_STAMPn access issue on FlexCAN instances with different number of MBs.
  - Fixed bit field MBDSR1-3 access issue on FlexCAN instances with different number of MBs.
- Improvements
  - Unified following API as same parameter and return value type:
    * FLEXCAN_GetMbStatusFlags
    * FLEXCAN_ClearMbStatusFlags
    * FLEXCAN_EnableMbInterrupts
    * FLEXCAN_DisableMbInterrupts
  - Add workaround for ERR050443 and ERR052403.
  - Update message buffer read process in API FLEXCAN_ReadRxMb and FLEXCAN_ReadFDRxMb to make critical section as short as possible.
  - Simplify API FLEXCAN_DriverDataIRQHandler implementation by remove parameter type.

[2.14.0]

- Improvements
  - Support external time tick feature.
  - Support high resolution timestamp feature.
  - Enter Freeze Mode first when enter Disable Mode on some platform.
  - Add feature macro for Pretended Networking because some FlexCAN instance do not have this feature.
  - Add feature macro for enhanced Rx FIFO because some FlexCAN instance do not have this feature.
  - Add new FlexCAN IRQ Handler FLEXCAN_DriverDataIRQHandler and FLEXCAN_DriverEventIRQHandler. Thses IRQ Handlers are used on soc which FlexCAN interrupts are grouped by specific function and assigned to different vector.
  - Update macro FLEXCAN_WAKE_UP_FLAG and FLEXCAN_PNWAKE_UP_FLAG to simplify code.
  - Replace macro FSL_FEATURE_FLEXCAN_HAS_NO_WAKMSK_SUPPORT with FSL_FEATURE_FLEXCAN_HAS_NO_SLFWAK_SUPPORT.
  - Replace macro FSL_FEATURE_FLEXCAN_HAS_NO_WAKSRC_SUPPORT with FSL_FEATURE_FLEXCAN_HAS_GLITCH_FILTER.
- Bug Fixes
  - Fixed wrong interrupt and status flag helper macro in enumeration _flexcan_flags and API FLEXCAN_DisableInterrupts.
  - Fixed interrupt flag helper macro typo issue.
  - Remove flags which will are unassociated with interrupt in macro FLEXCAN_MEMORY_ERROR_INT_FLAG.
  - Remove flags which will are unassociated with interrupt in macro FLEXCAN_ERROR_AND_STATUS_INT_FLAG.

- Fixed array out-of-bounds access when read enhanced Rx FIFO.

[2.13.1]

- Improvements
  - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.13.0]

- Improvements
  - Support payload endianness selection feature.

[2.12.0]

- Improvements
  - Support automatic Remote Response feature.
  - Add API FLEXCAN_SetRemoteResponseMbConfig() to configure automatic Remote Response mailbox.

[2.11.8]

- Improvements
  - Synchronize flexcan driver update on s32z platform.

[2.11.7]

- Bug Fixes
  - Fixed FLEXCAN_TransferReceiveEnhancedFifoEDMA() compatibility with edma5.

[2.11.6]

- Bug Fixes
  - Fixed ERRATA_9595 FLEXCAN_EnterFreezeMode() may result to bus fault on some platform.

[2.11.5]

- Bug Fixes
  - Fixed flexcan_memset() crash under high optimization compilation.

[2.11.4]

- Improvements
  - Update CANFD max bitrate to 10Mbps on MCXNx3x and MCXNx4x.
  - Release peripheral from reset if necessary in init function.

[2.11.3]

- Bug Fixes
  - Fixed FLEXCAN_TransferReceiveEnhancedFifoEDMA() compile error with DMA3.

[2.11.2]

- Bug Fixes
  - Fixed bug that timestamp in flexcan_handle_t not updated when RX overflow happens.

[2.11.1]

- Bug Fixes
  - Fixed violations of the MISRA C-2012 rules 10.1.

[2.11.0]

- Bug Fixes
  - Fixed wrong base address argument in FLEXCAN2 IRQ Handler.
- Improvements
  - Add API to determine if the instance supports CAN FD mode at run time.

[2.10.1]

- Bug Fixes
  - Fixed HIS CCM issue.
  - Fixed RTOS issue by adding protection to read-modify-write operations on interrupt enable/disable API.

[2.10.0]

- Improvements
  - Update driver to make it able to support devices which has more than 64 8bytes MBs.
  - Update CAN FD transfer APIs to make them set/get edl bit according to frame content, which can make them compatible with classic CAN.

[2.9.2]

- Bug Fixes
  - Fixed the issue that FLEXCAN_CheckUnhandleInterruptEvents() can't detecting the exist enhanced RX FIFO interrupt status.
  - Fixed the issue that FLEXCAN_ReadPNWakeUpMB() does not return fail even no existing valid wake-up frame.
  - Fixed the issue that FLEXCAN_ReadEnhancedRxFifo() may clear bits other than the data available bit.
  - Fixed violations of the MISRA C-2012 rules 10.4, 10.8.
- Improvements

  - Return kStatus_FLEXCAN_RxFifoDisabled instead of kStatus_Fail when read FIFO fail during IRQ handler.
  - Remove unreachable code from timing calculates APIs.
  - Update Enhanced Rx FIFO handler to make it deal with underflow/overflow status first.

[2.9.1]

- Bug Fixes
  - Fixed the issue that FLEXCAN_TransferReceiveEnhancedFifoBlocking() API clearing Fifo data available flag more than once.
  - Fixed the issue that entering FLEXCAN_SubHandlerForEhancedRxFifo() even if Enhanced Rx fifo interrupts are not enabled.
  - Fixed the issue that FLEXCAN_TransferReceiveEnhancedFifoEDMA() update handle even if previous Rx FIFO receive not finished.
  - Fixed the issue that FLEXCAN_SetEnhancedRxFifoConfig() not configure the ERFCR[NFE] bits to the correct value.
  - Fixed the issue that FLEXCAN_ReceiveFifoEDMACallback() can't differentiate between Rx fifo and enhanced rx fifo.
  - Fixed the issue that FLEXCAN_TransferHandleIRQ() can't report Legacy Rx FIFO warning status.

[2.9.0]

- Improvements
- Add public set bit rate API to make driver easier to use.
- Update Legacy Rx FIFO transfer APIs to make it support received multiple frames during one API call.
- Optimized FLEXCAN_SubHandlerForDataTransfered() API in interrupt handling to reduce the probability of packet loss.

[2.8.7]

- Improvements
- Initialized the EDMA configuration structure in the FLEXCAN EDMA driver.

[2.8.6]

- Bug Fixes
- Fix Coverity overrun issues in fsl_flexcan_edma driver.

[2.8.5]

- Improvements
  - Make driver aarch64 compatible.

[2.8.4]

- Bug Fixes
  - Fixed FlexCan_Errata_6032 to disable all interrupts.

[2.8.3]

- Bug Fixes
  - Fixed an issue with the FLEXCAN_EnableInterrupts and FLEXCAN_DisableInterrupts interrupt enable bits in the CTRL1 register.

[2.8.2]

- Bug Fixes
  - Fixed errors in timing calculations and simplify the calculation process.
  - Fixed issue of CBT and FDCBT register may write failure.

[2.8.1]

- Bug Fixes
  - Fixed the issue of CAN FD three sampling points.
  - Added macro to support the devices that no MCR[SUPV] bit.
  - Remove unnecessary clear WMB operations.

[2.8.0]

- Improvements
  - Update config configuration.
    - Added enableSupervisorMode member to support enable/disable Supervisor mode.
  - Simplified the algorithm in CAN FD improved timing APIs.

[2.7.1]

- Bug Fixes
  - Fixed violations of the MISRA C-2012 rules 10.3, 10.7.

[2.7.0]

- Improvements
  - Update config configuration.
    - Added enablePretendedeNetworking member to support enable/disable Pretended Networking feature.
    - Added enableTransceiverDelayMeasure member to support enable/disable Transceiver Delay MeasurementPretended feature.
    - Added bitRate/bitRateFD member to work as baudRate/baudRateFD member union.
  - Rename all “baud” in code or comments to “bit” to align with the CAN spec.

  - Added Pretended Networking mode related APIs.
    * FLEXCAN_SetPNConfig
    * FLEXCAN_GetPNMatchCount
    * FLEXCAN_ReadPNWakeUpMB
  - Added support for Enhanced Rx FIFO.
  - Removed independent memory error interrupt/status APIs and put all interrupt/status control operation into FLEXCAN_EnableInterrupts/FLEXCAN_DisableInterrupts and FLEXCAN_GetStatusFlags/FLEXCAN_ClearStatusFlags APIs.
  - Update improved timing APIs to make it calculate improved timing according to CiA doc recommended.
    * FLEXCAN_CalculateImprovedTimingValues.
    * FLEXCAN_FDCalculateImprovedTimingValues.
  - Update FLEXCAN_SetBitRate/FLEXCAN_SetFDBitRate to added the use of enhanced timing registers.

[2.6.2]

- Improvements
  - Add CANFD frame data length enumeration.

[2.6.1]

- Bug Fixes
  - Fixed the issue of not fully initializing memory in FLEXCAN_Reset() API.

[2.6.0]

- Improvements
  - Enable CANFD ISO mode in FLEXCAN_FDInit API.
  - Enable the transceiver delay compensation feature when enable FD operation and set bitrate switch.
  - Implementation memory error control in FLEXCAN_Init API.
  - Improve FLEXCAN_FDCalculateImprovedTimingValues API to get same value for FPRESDIV and PRESDIV.
  - Added memory error configuration for user.
    * enableMemoryErrorControl
    * enableNonCorrectableErrorEnterFreeze
  - Added memory error related APIs.
    * FLEXCAN_GetMemoryErrorReportStatus
    * FLEXCAN_GetMemoryErrorStatusFlags
    * FLEXCAN_ClearMemoryErrorStatusFlags
    * FLEXCAN_EnableMemoryErrorInterrupts
    * FLEXCAN_DisableMemoryErrorInterrupts
- Bug Fixes

  - Fixed the issue of sent duff CAN frame after call FLEXCAN_FDInit() API.

[2.5.2]

- Bug Fixes
  - Fixed the code error issue and simplified the algorithm in improved timing APIs.
    - The bit field in CTRL1 register couldn't calculate higher ideal SP, we set it as the lowest one(75%)
      - FLEXCAN_CalculateImprovedTimingValues
      - FLEXCAN_FDCalculateImprovedTimingValues
  - Fixed MISRA-C 2012 Rule 17.7 and 14.4.
- Improvements
  - Pass EsrStatus to callback function when kStatus_FLEXCAN_ErrorStatus is comming.

[2.5.1]

- Bug Fixes
  - Fixed the non-divisible case in improved timing APIs.
    - FLEXCAN_CalculateImprovedTimingValues
    - FLEXCAN_FDCalculateImprovedTimingValues

[2.5.0]

- Bug Fixes
  - MISRA C-2012 issue check.
    - Fixed rules, containing: rule-10.1, rule-10.3, rule-10.4, rule-10.7, rule-10.8, rule-11.8, rule-12.2, rule-13.4, rule-14.4, rule-15.5, rule-15.6, rule-15.7, rule-16.4, rule-17.3, rule-5.8, rule-8.3, rule-8.5.
  - Fixed the issue that API FLEXCAN_SetFDRxMbConfig lacks inactive message buff.
  - Fixed the issue of Pa082 warning.
  - Fixed the issue of dead lock in the function of interruption handler.
  - Fixed the issue of Legacy Rx Fifo EDMA transfer data fail in evkmimxrt1060 and evkmimxrt1064.
  - Fixed the issue of setting CANFD Bit Rate Switch.
  - Fixed the issue of operating unknown pointer risk.
    - when used the pointer "handle->mbFrameBuf[mbIdx]" to update the timestamp in a short-live TX frame, the frame pointer became as unknown, the action of operating it would result in program stack destroyed.
  - Added assert to check current CAN clock source affected by other clock gates in current device.
    - In some chips, CAN clock sources could be selected by CCM. But for some clock sources affected by other clock gates, if user insisted on using that clock source, they had to open these gates at the same time. However, they should take into consideration the power consumption issue at system level. In RT10xx chips, CAN clock source 2 was affected by the clock gate of lpuart1. ERRATA ID: (ERR050235 in CCM).

- Improvements
  - Implementation for new FLEXCAN with ECC feature able to exit Freeze mode.
  - Optimized the function of interruption handler.
  - Added two APIs for FLEXCAN EDMA driver.
    - FLEXCAN_PrepareTransfConfiguration
    - FLEXCAN_StartTransferDatafromRxFIFO
  - Added new API for FLEXCAN driver.
    - FLEXCAN_GetTimeStamp
      - For TX non-blocking API, we wrote the frame into mailbox only, so no need to register TX frame address to the pointer, and the timestamp could be updated into the new global variable handle->timestamp[mbIdx], the FLEXCAN driver provided a new API for user to get it by handle and index number after TX DONE Success.
    - FLEXCAN_EnterFreezeMode
    - FLEXCAN_ExitFreezeMode
  - Added new configuration for user.
    - disableSelfReception
    - enableListenOnlyMode
  - Renamed the two clock source enum macros based on CLKSRC bit field value directly.
    - The CLKSRC bit value had no property about Oscillator or Peripheral type in lots of devices, it acted as two different clock input source only, but the legacy enum macros name contained such property, that misled user to select incorrect CAN clock source.
  - Created two new enum macros for the FLEXCAN driver.
    - kFLEXCAN_ClkSrc0
    - kFLEXCAN_ClkSrc1
  - Deprecated two legacy enum macros for the FLEXCAN driver.
    - kFLEXCAN_ClkSrcOsc
    - kFLEXCAN_ClkSrcPeri
  - Changed the process flow for Remote request frame response..
    - Created a new enum macro for the FLEXCAN driver.
      - kStatus_FLEXCAN_RxRemote
  - Changed the process flow for kFLEXCAN_StateRxRemote state in the interrupt handler.
    - Should the TX frame not register to the pointer of frame handle, interrupt handler would not be able to read the remote response frame from the mail box to ram, so user should read the frame by manual from mail box after a complete remote frame transfer.

[2.4.0]

- Bug Fixes
  - MISRA C-2012 issue check.

    * Fixed rules, containing: rule-12.1, rule-17.7, rule-16.4, rule-11.9, rule-8.4, rule-14.4, rule-10.8, rule-10.4, rule-10.3, rule-10.7, rule-10.1, rule-11.6, rule-13.5, rule-11.3, rule-8.3, rule-12.2 and rule-16.1.
  - Fixed the issue that CANFD transfer data fail when bus baudrate is 30Khz.
  - Fixed the issue that ERR009595 does not folllow the ERRATA document.
  - Fixed code error for ERR006032 work around solution.
  - Fixed the Coverity issue of BAD_SHIFT in FLEXCAN.
  - Fixed the Repo build warning issue for variable without initial.
- Improvements
  - Fixed the run fail issue of FlexCAN RemoteRequest UT Case.
  - Implementation all TX and RX transfering Timestamp used in FlexCAN demos.
  - Fixed the issue of UT Test Fail for CANFD payload size changed from 64BperMB to 8PerMB.
  - Implementation for improved timing API by baud rate.

[2.3.2]

- Improvements
  - Implementation for ERR005959.
  - Implementation for ERR005829.
  - Implementation for ERR006032.

[2.3.1]

- Bug Fixes
  - Added correct handle when kStatus_FLEXCAN_TxSwitchToRx is comming.

[2.3.0]

- Improvements
  - Added self-wakeup support for STOP mode in the interrupt handling.

[2.2.3]

- Bug Fixes
  - Fixed the issue of CANFD data phase's bit rate not set as expected.

[2.2.2]

- Improvements
  - Added a time stamp feature and enable it in the interrupt_transfer example.

[2.2.1]

- Improvements
  - Separated CANFD initialization API.
  - In the interrupt handling, fix the issue that the user cannot use the normal CAN API when with an FD.

[2.2.0]

- Improvements
  - Added FSL_FEATURE_FLEXCAN_HAS_SUPPORT_ENGINE_CLK_SEL_REMOVE feature to support SoCs without CAN Engine Clock selection in FlexCAN module.
  - Added FlexCAN Serial Clock Operation to support i.MX SoCs.

[2.1.0]

- Bug Fixes
  - Corrected the spelling error in the function name FLEXCAN_XXX().
  - Moved Freeze Enable/Disable setting from FLEXCAN_Enter/ExitFreezeMode() to FLEXCAN_Init().
  - Corrected wrong helper macro values.
- Improvements
  - Hid FLEXCAN_Reset() from user.
  - Used NDEBUG macro to wrap FLEXCAN_IsMbOccupied() function instead of DEBUG macro.

[2.0.0]

- Initial version.

FLEXCAN_EDMA

[2.12.1]

- Bug Fixes
  - Fixed violations of the MISRA C-2012 rules 18.1.

[2.12.0]

- Improvements
  - Support high resolution timestamp feature in enhanced Rx FIFO EDMA.
  - Add feature macro for enhanced Rx FIFO because some FlexCAN instance do not have this feature.
- Bug Fixes
  - Fixed array out-of-bounds access when read enhanced Rx FIFO in EDMA.

[2.11.7]

- Refer FLEXCAN driver change log 2.7.0 to 2.11.7

FLEXIO

[2.3.0]

- Improvements
  - Supported platforms which don't have DOZE mode control.
  - Added more pin control functions.

[2.2.3]

- Improvements
  - Adapter the FLEXIO driver to platforms which don't have system level interrupt controller, such as NVIC.

[2.2.2]

- Improvements
  - Release peripheral from reset if necessary in init function.

[2.2.1]

- Improvements
  - Added doxygen index parameter comment in FLEXIO_SetClockMode.

[2.2.0]

- New Features
  - Added new APIs to support FlexIO pin register.

[2.1.0]

- Improvements
  - Added API FLEXIO_SetClockMode to set flexio channel counter and source clock.

[2.0.4]

- Bug Fixes
  - Fixed MISRA 8.4 issues.

[2.0.3]

- Bug Fixes
  - Fixed MISRA 10.4 issues.

[2.0.2]

- Improvements
  - Split FLEXIO component which combines all flexio/flexio_uart/flexio_i2c/flexio_i2s drivers into several components: FlexIO component, flexio_uart component, flexio_i2c_master component, and flexio_i2s component.
- Bug Fixes
  - Fixed MISRA issues
    - Fixed rules 10.1, 10.3, 10.4, 10.7, 11.6, 11.9, 14.4, 17.7.

[2.0.1]

- Bug Fixes
  - Fixed the dozen mode configuration error in FLEXIO_Init API. For enableInDoze = true, the configuration should be 0; for enableInDoze = false, the configuration should be 1.

FLEXIO_I2C

[2.6.2]

- Improvements
  - Added timeout for while loop in FLEXIO_I2C_MasterTransferBlocking().
- Bug Fixes
  - Fixed build issues related to I2C_RETRY_TIMES.

[2.6.1]

- Bug Fixes
  - Fixed coverity issues

[2.6.0]

- Improvements
  - Supported platforms which don't have DOZE mode control.

[2.5.1]

- Improvements
  - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.5.0]

- Improvements
  - Split some functions, fixed CCM problem in file fsl_flexio_i2c_master.c.

[2.4.0]

- Improvements
  - Added delay of 1 clock cycle in FLEXIO_I2C_MasterTransferRunStateMachine to ensure that bus would be idle before next transfer if master is nacked.
  - Fixed issue that the restart setup time is less than the time in I2C spec by adding delay of 1 clock cycle before restart signal.

[2.3.0]

- Improvements
  - Used 3 timers instead of 2 to support transfer which is more than 14 bytes in single transfer.
  - Improved FLEXIO_I2C_MasterTransferGetCount so that the API can check whether the transfer is still in progress.
- Bug Fixes
  - Fixed MISRA 10.4 issues.

[2.2.0]

- New Features
  - Added timeout mechanism when waiting certain state in transfer API.
  - Added an API for checking bus pin status.
- Bug Fixes
  - Fixed COVERITY issue of useless call in FLEXIO_I2C_MasterTransferRunStateMachine.
  - Fixed MISRA issues
    - Fixed rules 10.1, 10.3, 10.4, 10.7, 11.6, 11.9, 14.4, 17.7.
  - Added codes in FLEXIO_I2C_MasterTransferCreateHandle to clear pending NVIC IRQ, disable internal IRQs before enabling NVIC IRQ.
  - Modified code so that during master's nonblocking transfer the start and slave address are sent after interrupts being enabled, in order to avoid potential issue of sending the start and slave address twice.

[2.1.7]

- Bug Fixes
  - Fixed the issue that FLEXIO_I2C_MasterTransferBlocking did not wait for STOP bit sent.
  - Fixed COVERITY issue of useless call in FLEXIO_I2C_MasterTransferRunStateMachine.
  - Fixed the issue that I2C master did not check whether bus was busy before transfer.

[2.1.6]

- Bug Fixes
  - Fixed the issue that I2C Master transfer APIs(blocking/non-blocking) did not support the situation of master transfer with subaddress and transfer data size being zero, which means no data followed the subaddress.

[2.1.5]

- Improvements
  - Unified component full name to FLEXIO I2C Driver.

[2.1.4]

- Bug Fixes
  - The following modifications support FlexIO using multiple instances:
    * Removed FLEXIO_Reset API in module Init APIs.
    * Updated module Deinit APIs to reset the shifter/timer config instead of disabling module/clock.
    * Updated module Enable APIs to only support enable operation.

[2.1.3]

- Improvements
  - Changed the prototype of FLEXIO_I2C_MasterInit to return kStatus_Success if initialized successfully or to return kStatus_InvalidArgument if "(srcClock_Hz / masterConfig->baudRate_Bps) / 2 - 1" exceeds 0xFFU.

[2.1.2]

- Bug Fixes
  - Fixed the FLEXIO I2C issue where the master could not receive data from I2C slave in high baudrate.
  - Fixed the FLEXIO I2C issue where the master could not receive NAK when master sent non-existent addr.
  - Fixed the FLEXIO I2C issue where the master could not get transfer count successfully.
  - Fixed the FLEXIO I2C issue where the master could not receive data successfully when sending data first.
  - Fixed the Dozen mode configuration error in FLEXIO_I2C_MasterInit API. For enableInDoze = true, the configuration should be 0; for enableInDoze = false, the configuration should be 1.
  - Fixed the issue that FLEXIO_I2C_MasterTransferBlocking API called FLEXIO_I2C_MasterTransferCreateHandle, which lead to the s_flexioHandle/s_flexioIsr/s_flexioType variable being written. Then, if calling FLEXIO_I2C_MasterTransferBlocking API multiple times, the s_flexioHandle/s_flexioIsr/s_flexioType variable would not be written any more due to it being out of range. This lead to the following situation: NonBlocking transfer APIs could not work due to the fail of register IRQ.

[2.1.1]

- Bug Fixes
  - Implemented the FLEXIO_I2C_MasterTransferBlocking API which is defined in header file but has no implementation in the C file.

[2.1.0]

- New Features
  - Added Transfer prefix in transactional APIs.
  - Added transferSize in handle structure to record the transfer size.

FLEXIO_I2S

[2.2.2]

- Bug Fixes
  - Fixed violations of MISRA C-2012 rule 12.4.

[2.2.1]

- Improvements
  - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.2.0]

- New Features
  - Added timeout mechanism when waiting certain state in transfer API.
- Bug Fixes
  - Fixed IAR Pa082 warnings.
  - Fixed violations of the MISRA C-2012 rules 10.4, 14.4, 11.8, 11.9, 10.1, 17.7, 11.6, 10.3, 10.7.

[2.1.6]

- Bug Fixes
  - Added reset flexio before flexio i2s init to make sure flexio status is normal.

[2.1.5]

- Bug Fixes
  - Fixed the issue that I2S driver used hard code for bitwidth setting.

[2.1.4]

- Improvements
  - Unified component's full name to FLEXIO I2S (DMA/EDMA) driver.

[2.1.3]

- Bug Fixes
  - The following modifications support FLEXIO using multiple instances:
    * Removed FLEXIO_Reset API in module Init APIs.
    * Updated module Deinit APIs to reset the shifter/timer config instead of disabling module/clock.
    * Updated module Enable APIs to only support enable operation.

[2.1.2]

- New Features
  - Added configure items for all pin polarity and data valid polarity.
  - Added default configure for pin polarity and data valid polarity.

[2.1.1]

- Bug Fixes
  - Fixed FlexIO I2S RX data read error and eDMA address error.
  - Fixed FlexIO I2S slave timer compare setting error.

[2.1.0]

- New Features
  - Added Transfer prefix in transactional APIs.
  - Added transferSize in handle structure to record the transfer size.

FLEXIO_SPI

[2.4.3]

- Improvements
  - Make SPI_RETRY_TIMES configurable by CONFIG_SPI_RETRY_TIMES.

[2.4.2]

- Bug Fixes
  - Fixed FLEXIO_SPI_MasterTransferBlocking and FLEXIO_SPI_MasterTransferNonBlocking issue in CS continuous mode, the CS might not be continuous.

[2.4.1]

- Bug Fixes
  - Fixed coverity issues

[2.4.0]

- Improvements
  - Supported platforms which don't have DOZE mode control.

[2.3.5]

- Improvements
  - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.3.4]

- Bug Fixes
  - Fixed the txData from void * to const void * in transmit API

[2.3.3]

- Bugfixes
  - Fixed cs-continuous mode.

[2.3.2]

- Improvements
  - Changed FLEXIO_SPI_DUMMYDATA to 0x00.

[2.3.1]

- Bugfixes
  - Fixed IRQ SHIFTBUF overrun issue when one FLEXIO instance used as multiple SPIs.

[2.3.0]

- New Features
  - Supported FLEXIO_SPI slave transfer with continuous master CS signal and CPHA=0.
  - Supported FLEXIO_SPI master transfer with continuous CS signal.
  - Support 32 bit transfer width.
- Bug Fixes
  - Fixed wrong timer compare configuration for dma/edma transfer.
  - Fixed wrong byte order of rx data if transfer width is 16 bit, since the we use shifter buffer bit swapped/byte swapped register to read in received data, so the high byte should be read from the high bits of the register when MSB.

[2.2.1]

- Bug Fixes
  - Fixed bug in FLEXIO_SPI_MasterTransferAbortEDMA that when aborting EDMA transfer EDMA_AbortTransfer should be used rather than EDMA_StopTransfer.

[2.2.0]

- Improvements
  - Added timeout mechanism when waiting certain states in transfer driver.
- Bug Fixes
  - Fixed MISRA 10.4 issues.
  - Added codes in FLEXIO_SPI_MasterTransferCreateHandle and FLEXIO_SPI_SlaveTransferCreateHandle to clear pending NVIC IRQ before enabling NVIC IRQ, to fix issue of pending IRQ interfering the on-going process.

[2.1.3]

- Improvements
  - Unified component full name to FLEXIO SPI(DMA/EDMA) Driver.
- Bug Fixes
  - Fixed MISRA issues
    - Fixed rules 10.1, 10.3, 10.4, 10.7, 11.6, 11.9, 14.4, 17.7.

[2.1.2]

- Bug Fixes
  - The following modification support FlexIO using multiple instances:
    - Removed FLEXIO_Reset API in module Init APIs.
    - Updated module Deinit APIs to reset the shifter/timer config instead of disabling module/clock.
    - Updated module Enable APIs to only support enable operation.

[2.1.1]

- Bug Fixes
  - Fixed bug where FLEXIO SPI transfer data is in 16 bit per frame mode with eDMA.
  - Fixed bug when FLEXIO SPI works in eDMA and interrupt mode with 16-bit per frame and Lsbfirst.
  - Fixed the Dozen mode configuration error in FLEXIO_SPI_MasterInit/FLEXIO_SPI_SlaveInit API. For enableInDoze = true, the configuration should be 0; for enableInDoze = false, the configuration should be 1.
- Improvements
  - Added #ifndef/#endif to allow users to change the default TX value at compile time.

[2.1.0]

- New Features
  - Added Transfer prefix in transactional APIs.
  - Added transferSize in handle structure to record the transfer size.
- Bug Fixes

  - Fixed the error register address return for 16-bit data write in FLEXIO_SPI_GetTxDataRegisterAddress.
  - Provided independent IRQHandler/transfer APIs for Master and slave to fix the baudrate limit issue.

FLEXIO_UART

[2.6.4]

- Improvements
  - Make UART_RETRY_TIMES configurable by CONFIG_UART_RETRY_TIMES.

[2.6.3]

- Bug Fixes
  - Fixed coverity issues

[2.6.2]

- Bug Fixes
  - Fixed coverity issues

[2.6.1]

- Improvements
  - Improve baudrate calculation method, to support higher frequency FlexIO clock source.

[2.6.0]

- Improvements
  - Supported platforms which don't have DOZE mode control.

[2.5.1]

- Improvements
  - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.5.0]

- Improvements
  - Added API FLEXIO_UART_FlushShifters to flush UART fifo.

[2.4.0]

- Improvements
  - Use separate data for TX and RX in flexio_uart_transfer_t.
- Bug Fixes
  - Fixed bug that when ring buffer is used, if some data is received in ring buffer first before calling FLEXIO_UART_TransferReceiveNonBlocking, the received data count returned by FLEXIO_UART_TransferGetReceiveCount is wrong.

[2.3.0]

- Improvements
  - Added check for baud rate's accuracy that returns kStatus_FLEXIO_UART_BaudrateNotSupport when the best achieved baud rate is not within 3% error of configured baud rate.
- Bug Fixes
  - Added codes in FLEXIO_UART_TransferCreateHandle to clear pending NVIC IRQ before enabling NVIC IRQ, to fix issue of pending IRQ interfering the on-going process.

[2.2.0]

- Improvements
  - Added timeout mechanism when waiting for certain states in transfer driver.
- Bug Fixes
  - Fixed MISRA 10.4 issues.

[2.1.6]

- Bug Fixes
  - Fixed IAR Pa082 warnings.
  - Fixed MISRA issues
    - Fixed rules 10.1, 10.3, 10.4, 10.7, 11.6, 11.9, 14.4, 17.7.

[2.1.5]

- Improvements
  - Triggered user callback after all the data in ringbuffer were received in FLEXIO_UART_TransferReceiveNonBlocking.

[2.1.4]

- Improvements
  - Unified component full name to FLEXIO UART(DMA/EDMA) Driver.

[2.1.3]

- Bug Fixes
  - The following modifications support FLEXIO using multiple instances:
    - Removed FLEXIO_Reset API in module Init APIs.
    - Updated module Deinit APIs to reset the shifter/timer configuration instead of disabling module and clock.
    - Updated module Enable APIs to only support enable operation.

[2.1.2]

- Bug Fixes
  - Fixed the transfer count calculation issue in FLEXIO_UART_TransferGetReceiveCount, FLEXIO_UART_TransferGetSendCount, FLEXIO_UART_TransferGetReceiveCountDMA, FLEXIO_UART_TransferGetSendCountDMA, FLEXIO_UART_TransferGetReceiveCountEDMA and FLEXIO_UART_TransferGetSendCountEDMA.
  - Fixed the Dozen mode configuration error in FLEXIO_UART_Init API. For enableInDoze = true, the configuration should be 0; for enableInDoze = false, the configuration should be 1.
  - Added code to report errors if the user sets a too-low-baudrate which FLEXIO cannot reach.
  - Disabled FLEXIO_UART receive interrupt instead of all NVICs when reading data from ring buffer. If ring buffer is used, receive nonblocking will disable all NVIC interrupts to protect the ring buffer. This had negative effects on other IPs using interrupt.

[2.1.1]

- Bug Fixes
  - Changed the API name FLEXIO_UART_StopRingBuffer to FLEXIO_UART_TransferStopRingBuffer to align with the definition in C file.

[2.1.0]

- New Features
  - Added Transfer prefix in transactional APIs.
  - Added txSize/rxSize in handle structure to record the transfer size.
- Bug Fixes
  - Added an error handle to handle the situation that data count is zero or data buffer is NULL.

FLEXIO_UART_EDMA

[2.3.1]

- Bug Fixes
  - Fixed violations of the MISRA C-2012 rules.

[2.3.0]

- Refer FLEXIO_UART driver change log to 2.3.0

GPIO

[2.8.3]

- Bug Fixes
  - Fixed violations of the MISRA C-2012 Rule 10.1, 5.7.

[2.8.2]

- Bug Fixes
  - Fixed COVERITY issue that GPIO_GetInstance could return clock array overflow values due to GPIO base and clock being out of sync.

[2.8.1]

- Bug Fixes
  - Fixed CERT INT31-C issues.

[2.8.0]

- Improvements
  - Add API GPIO_PortInit/GPIO_PortDeinit to set GPIO clock enable and releasing GPIO reset.

[2.8.0]

- Improvements
  - Add API GPIO_PortInit/GPIO_PortDeinit to set GPIO clock enable and releasing GPIO reset.
  - Remove support for API GPIO_GetPinsDMARequestFlags with GPIO_ISFR_COUNT <= 1.

[2.7.3]

- Improvements
  - Release peripheral from reset if necessary in init function.

[2.7.2]

- New Features
  - Support devices without PORT module.

[2.7.1]

- Bug Fixes
  - Fixed MISRA C-2012 rule 10.4 issues in GPIO_GpioGetInterruptChannelFlags() function and GPIO_GpioClearInterruptChannelFlags() function.

[2.7.0]

- New Features
  - Added API to support Interrupt select (IRQS) bitfield.

[2.6.0]

- New Features
  - Added API to get GPIO version information.
  - Added API to control a pin for general purpose input.
  - Added some APIs to control pin in secure and previliege status.

[2.5.3]

- Bug Fixes
  - Correct the feature macro typo: FSL_FEATURE_GPIO_HAS_NO_INDEP_OUTPUT_CONTORL.

[2.5.2]

- Improvements
  - Improved GPIO_PortSet/GPIO_PortClear/GPIO_PortToggle functions to support devices without Set/Clear/Toggle registers.

[2.5.1]

- Bug Fixes
  - Fixed wrong macro definition.
  - Fixed MISRA C-2012 rule issues in the FGPIO_CheckAttributeBytes() function.
  - Defined the new macro to separate the scene when the width of registers is different.
  - Removed some redundant macros.
- New Features
  - Added some APIs to get/clear the interrupt status flag when the port doesn't control pins' interrupt.

[2.4.1]

- Improvements
  - Improved GPIO_CheckAttributeBytes() function to support 8 bits width GACR register.

[2.4.0]

- Improvements
  - API interface added:
    - New APIs were added to configure the GPIO interrupt clear settings.

[2.3.2]

- Bug Fixes
  - Fixed the issue for MISRA-2012 check.
    - Fixed rule 3.1, 10.1, 8.6, 10.6, and 10.3.

[2.3.1]

- Improvements
  - Removed deprecated APIs.

[2.3.0]

- New Features
  - Updated the driver code to adapt the case of interrupt configurations in GPIO module. New APIs were added to configure the GPIO interrupt settings if the module has this feature on it.

[2.2.1]

- Improvements
  - API interface changes:
    - Refined naming of APIs while keeping all original APIs by marking them as deprecated. The original APIs will be removed in next release. The main change is updating APIs with prefix of _PinXXX() and _PortXXX.

[2.1.1]

- Improvements
  - API interface changes:
    - Added an API for the check attribute bytes.

[2.1.0]

- Improvements
  - API interface changes:
    - Added “pins” or “pin” to some APIs’ names.
    - Renamed “_PinConfigure” to “GPIO_PinInit”.

I3C

[2.14.3]

- Improvements
  - Fixed Coverity CERT-C violations.
  - Used I3C_RSTS instead of I3C special feature macro.
  - Adapted the driver to support new platform.

[2.14.2]

- Improvements
  - Added timeout for ENTDAA process API.
  - Added build system macro to control the timeout setting.

[2.14.1]

- Improvements
  - Split the function I3C_MasterTransferBlocking to meet the HIS-CCM requirement.

[2.14.0]

- Improvements
  - Added the choice to set fast start header with push-pull speed when all targets addresses have MSB 0 instead of forcing to set it.
  - Deleted duplicated busy check in I3C_MasterStart function.

[2.13.1]

- Bug Fixes
  - Disabled Rx auto-termination in repeated start interrupt event while transfer API doesn't enable it.
  - Waited the completion event after loading all Tx data in Tx FIFO.
- Improvements
  - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.13.0]

- New features
  - Added the hot-join support for I3C bus initialization API.
- Bug Fixes
  - Set read termination with START at the same time in case unknown issue.
  - Set MCTRL[TYPE] as 0 for DDR force exit.
- Improvements
  - Added the API to reset device count assigned by ENTDAA.

  - Provided the method to set global macro I3C_MAX_DEVCNT to determine how many device addresses ENTDAA can allocate at one time.
  - Initialized target management static array based on instance number for the case that multiple instances are used at the same time.

[2.12.0]

- Improvements
  - Added the slow clock parameter for Controller initialization function to calculate accurate timeout.
- Bug Fixes
  - Fixed the issue that BAMATCH field can't be 0. BAMATCH should be 1 for 1MHz slow clock.

[2.11.1]

- Bug Fixes
  - Fixed the issue that interrupt API transmits extra byte when subaddress and data size are null.
  - Fixed the slow clock calculation issue.

[2.11.0]

- New features
  - Added the START/ReSTART SCL delay setting for the Soc which supports this feature.
- Bug Fixes
  - Fixed the issue that ENTDAA process waits Rx pending flag which causes problem when Rx watermark isn't 0. Just check the Rx FIFO count.

[2.10.8]

- Improvements
  - Support more instances.

[2.10.7]

- Improvements
  - Fixed the potential compile warning.

[2.10.6]

- New features
  - Added the I3C private read/write with 0x7E address as start.

[2.10.5]

- New features
  - Added I3C HDR-DDR transfer support.

[2.10.4]

- Improvements
  - Added one more option for master to not set RDTERM when doing I3C Common Command Code transfer.

[2.10.3]

- Improvements
  - Masked the slave IBI/MR/HJ request functions with feature macro.

[2.10.2]

- Bug Fixes
  - Added workaround for errata ERR051617: I3C working with I2C mode creates the unintended Repeated START before actual STOP on some platforms.

[2.10.1]

- Bug Fixes
  - Fixed the issue that DAA function doesn't wait until all Rx data is read out from FIFO after master control done flag is set.
  - Fixed the issue that DAA function could return directly although the disabled interrupts are not enabled back.

[2.10.0]

- New features
  - Added I3C extended IBI data support.

[2.9.0]

- Improvements
  - Added adaptive termination for master blocking transfer. Set termination with start signal when receiving bytes less than 256.

[2.8.2]

- Improvements
  - Fixed the build warning due to armgcc strict check.

[2.8.1]

- Bug Fixes
  - Fixed violations of the MISRA C-2012 rules 17.7.

[2.8.0]

- Improvements
  - Added API I3C_MasterProcessDAASpecifiedBaudrate for temporary baud rate adjustment when I3C master assigns dynamic address.

[2.7.1]

- Bug Fixes
  - Fixed the issue that I3C slave handle STOP event before finishing data transmission.

[2.7.0]

- Fixed the CCM problem in file fsl_i3c.c.
- Fixed the FSL_FEATURE_I3C_HAS_NO_SCONFIG_IDRAND usage issue in I3C_GetDefaultConfig and I3C_Init.

[2.6.0]

- Fixed the FSL_FEATURE_I3C_HAS_NO_SCONFIG_IDRAND usage issue in fsl_i3c.h.
- Changed some static functions in fsl_i3c.c as non-static and define the functions in fsl_i3c.h to make I3C DMA driver reuse:
  - I3C_GetIBIType
  - I3C_GetIBIAddress
  - I3C_SlaveCheckAndClearError
- Changed the handle pointer parameter in IRQ related funtions to void * type to make it reuse in I3C DMA driver.
- Added new API I3C_SlaveRequestIBIWithSingleData for slave to request single data byte, this API could be used regardless slave is working in non-blocking interrupt or non-blocking dma.
- Added new API I3C_MasterGetDeviceListAfterDAA for master application to get the device information list built up in DAA process.

[2.5.4]

- Improved I3C driver to avoid setting state twice in the SendCommandState of I3C_RunTransferStateMachine.
- Fixed MISRA violation of rule 20.9.
- Fixed the issue that I3C_MasterEmitRequest did not use Type I3C SDR.

[2.5.3]

- Updated driver for new feature FSL_FEATURE_I3C_HAS_NO_SCONFIG_BAMATCH and FSL_FEATURE_I3C_HAS_NO_SCONFIG_IDRAND.

[2.5.2]

- Updated driver for new feature FSL_FEATURE_I3C_HAS_NO_MERRWARN_TERM.
- Fixed the issue that call to I3C_MasterTransferBlocking API did not generate STOP signal when NAK status was returned.

[2.5.1]

- Improved the receive terminate size setting for interrupt transfer read, now it's set at beginning of transfer if the receive size is less than 256 bytes.

[2.5.0]

- Added new API I3C_MasterRepeatedStartWithRxSize to send repeated start signal with receive terminate size specified.
- Fixed the status used in I3C_RunTransferStateMachine, changed to use pending interrupts as status to be handled in the state machine.
- Fixed MISRA 2012 violation of rule 10.3, 10.7.

[2.4.0]

- Bug Fixes
  - Fixed kI3C_SlaveMatchedFlag interrupt is not properly handled in I3C_SlaveTransferHandleIRQ when it comes together with interrupt kI3C_SlaveBusStartFlag.
  - Fixed the inaccurate I2C baudrate calculation in I3C_MasterSetBaudRate.
  - Added new API I3C_MasterGetIBIRules to get registered IBI rules.
  - Added new variable isReadTerm in struct _i3c_master_handle for transfer state routine to check if MCTRL.RDTERM is configured for read transfer.
  - Changed to emit Auto IBI in transfer state routine for slave start flag assertion.
  - Fixed the slave maxWriteLength and maxReadLength does not be configured into SMAXLIMITS register issue.
  - Fixed incorrect state for IBI in I3C master interrupt transfer IRQ handle routine.
  - Added isHotJoin in i3c_slave_config_t to request hot-join event during slave init.

[2.3.2]

- Bug Fixes
  - Fixed violations of the MISRA C-2012 rules 8.4, 17.7.
  - Fixed incorrect HotJoin event index in I3C_GetIBIType.

[2.3.1]

- Bug Fixes
  - Fixed the issue that call of I3C_MasterTransferBlocking/I3C_MasterTransferNonBlocking fails for the case which receive length 1 byte of data.
  - Fixed the issue that STOP signal is not sent when NAK status is detected during execution of I3C_MasterTransferBlocking function.

[2.3.0]

- Improvements
  - Added I3C common driver APIs to initialize I3C with both master and slave configuration.

  - Updated I3C master transfer callback to function set structure to include callback invoke for IBI event and slave2master event.
  - Updated I3C master non-blocking transfer model and always enable the interrupts to be able to re-act to the slave start event and handle slave IBI.

[2.2.0]

- Bug Fixes
  - Fixed the issue that I3C transfer size limit to 255 bytes.

[2.1.2]

- Bug Fixes
  - Reset default hkeep value to kI3C_MasterHighKeeperNone in I3C_MasterGetDefaultConfig

[2.1.1]

- Bug Fixes
  - Fixed incorrect FIFO reset operation in I3C Master Transfer APIs.
  - Fixed i3c slave IRQ handler issue, slave transmit could be underrun because tx FIFO is not filled in time right after start flag detected.

[2.1.0]

- Added definitions and APIs for I3C slave functionality, updated previous I3C APIs to support I3C functionality.

[2.0.0]

- Initial version.

IMU

[2.2.1]

- Bug Fixes
  - Fix MISRA C-2012 Rule 14.3 violations.

[2.2.0]

- New Features
  - Added IMU_BUSY_POLL_COUNT parameter to prevent infinite polling loops in IMU operations.
  - Added timeout mechanism to all polling loops in IMU driver code.
- Improvements
  - Enhanced error handling in blocking functions to return timeout status.
  - Updated documentation to clarify timeout behavior and return values.

  - Added IMU_ERR_TIMEOUT error code for timeout conditions.

[2.1.1]

- Bug Fixes
  - Fix MISRA C-2012 violations.
  - Fixed IMU_GetStatusFlags bug that returns wrong RX FIFO status.

[2.1.0]

- Improvements:
  - Updated API prototype, remove CIU2_Type from parameters.

[2.0.0]

- Initial version.

LIN

[2.2.4]

- Improvements
  - Added Kconfig item LIN_LPUART_TRANSMISSION_COMPLETE_TIMEOUT, user can configure it to none zero value for the while loop to break after retrying LIN_LPUART_TRANSMISSION_COMPLETE_TIMEOUT times.

[2.2.3]

- Improvements
  - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.2.2]

- Bug Fixes
  - Fixed bug in LIN_LPUART_Init, while disabling the LPUART ERROR IRQ, the variable g_linLpuartErrIrqId should be used instead of g_linLpuartRxTxIrqId. This is a typo issue.

[2.2.1]

- Bug Fixes
  - Fixed bug in LIN_LPUART_GetTransmitStatus and LIN_LPUART_GetReceiveStatus, if the current event ID is timeout, then the timeout service has already been executed, LIN_TIMEOUT should be returned.
  - Fixed bug in LIN_LPUART_AutobaudTimerValEval, if the calculated MasterBaudrate is 0 which means the baudrate is not supported, then there is no need to update the baudrate configuration, just set the node to idle state.
  - Fixed issue in LIN_LPUART_CheckWakeupSignal, the check on the timerGetTimeIntervalCallback should be executed after the first falling edge rather than the second.

[2.2.0]

- Bug Fixes
  - Fixed bug that for lin slave once break field is detected before completing the last frame no error status is raised.

[2.1.1]

- Bug Fixes
  - Fixed the MISRA C-2012 issue.
    - Fixed rules, containing: rule-10.8, 10.3.
  - Fixed bug in LIN_MakeChecksumByte, when frame ID is 0x3D the PID should be 0x7D.

[2.1.0]

- Improvements
  - Updated the driver to support LIN stack.
    - For LIN_LPUART_Init() function, if run successfully, the return is LIN_SUCCESS instead of LIN_INITIALIZED.
    - Changed the LIN event ID name from "LIN_RECV_SYNC_OK" to "LIN_SYNC_OK".
    - Changed the LIN event ID name from "LIN_RECV_SYNV_ERROR" to "LIN_SYNV_ERROR".
    - Changed the definition macro name "LIN_MAKE_PARITY" to "MAKE_PARITY".
    - Changed the definition macro name "LIN_CHECK_PARITY" to "CHECK_PARITY".
- Bug Fixes
  - Fixed the MISRA C-2012 issue.
    - Fixed rules, containing: rule-17.7, rule-14.4, rule-11.9, rule-10.4, rule-10.3, rule-10.7, rule-10.1, rule-8.6, rule-8.4, rule-11.3, rule-10.8, and rule-16.1.

[2.0.1]

- Bug Fixes
  - Fixed bug where resting LPUART FIFO initializes the LPUART module.

[2.0.0]

- Initial version.

LPADC

[2.9.5]

- Improvements
  - Fix doxygen issue, grouping command should be balanced.

[2.9.4]

- Improvements
  - Update LPADC_GetDefaultConfig, change default conversionAverageMode value to: kLPADC_ConversionAverage128 for 3 bit width. kLPADC_ConversionAverage1024 for 4 bit width.

[2.9.3]

- Improvements
  - Add timeout for while loop code.

[2.9.2]

- Improvements
  - Fixed CERT-C issues.

[2.9.1]

- Bug Fixes
  - Fixed incorrect channel B FIFO selection logic.

[2.9.0]

- Bug Fixes
  - Add code to handle the case where GCC[GAIN_CAL] is a signed number.
  - Split LPADC_FinishAutoCalibration function into two functions.
  - Improved LPADC driver.

[2.8.4]

- Bug Fixes
  - Remove function 'LPADC_SetOffsetValue' assert statement, this statement may cause runtime errors in existing code.

[2.8.3]

- Bug Fixes
  - Fixed SDK lpadc driver examples compile issue, move condition 'commandId < ADC_CV_COUNT' to a more appropriate location.

[2.8.2]

- Bug Fixes
  - Fixed the violations of MISRA C-2012 rule 18.1, 10.3, 10.1 and 10.4.

[2.8.1]

- Bug Fixes
  - Fixed LPADC sample mode enum name mistake.

[2.8.0]

- Improvements
  - Release peripheral from reset if necessary in init function.
- Bug Fixes
  - Fixed function LPADC_GetConvResult() issue.
  - Fixed function LPADC_SetConvCommandConfig() bugs.

[2.7.2]

- Improvements
  - Use feature macros instead of header file macros.
- Bug Fixes
  - Fixed the violations of MISRA C-2012 rule 10.1, 10.3, 10.4 and 14.3.

[2.7.1]

- Improvements
  - Corrected descriptions of several functions.
  - Improved function LPADC_GetOffsetValue and LPADC_SetOffsetValue.
  - Revert changes of feature macros for lpadc.
  - Use feature macros instead of header file macros.
- Bug Fixes
  - Fixed the violations of MISRA C-2012 rule 10.8.
  - Fixed the violations of MISRA C-2012 rule 10.1, 10.3, 10.4 and 14.3.

[2.7.0]

- Improvements
  - Added supports of CFG2 register.
  - Removed some useless macros.

[2.6.2]

- Bug Fixes
  - Fixed the violations of MISRA C-2012 rules.
  - Fixed LPADC driver code compile error issue.

[2.6.1]

- Improvements
  - Updated the use of macros in the driver code.

[2.6.0]

- Improvements
  - Added the API LPADC_SetOffset12BitValue() to configure 12bit ADC conversion offset trim value manually.
  - Added the API LPADC_SetOffset16BitValue() to configure 16bit ADC conversion offset trim value manually.
  - Added API to set offset calibration mode.
  - Added configuration of alternate channel.
  - Updated auto calibration API and added calibration value conversion API.
- New feature
  - Added API LPADC_EnableHardwareTriggerCommandSelection() to enable trigger commands controlled by ADC_ETC.
  - Updated LPADC_DoAutoCalibration() to allow doing something else before the ADC initialization to be totally complete. Enhance initialization duration time of the ADC.
  - Added two new APIs to get/set calibration value.

[2.5.2]

- Improvements
  - Added while loop, LPADC_GetConvResult() will return only when the FIFO will not be empty.

[2.5.1]

- Bug Fixes
  - Fixed some typos in Lpadc driver comments.

[2.5.0]

- Improvements
  - Added missing items to enable trigger interrupts.

[2.4.0]

- New features
  - Added APIs to get/clear trigger status flags.

[2.3.0]

- Improvements
  - Removed LPADC_MeasureTemperature() function for the LPADC supports different temperature sensor calculation equations.

[2.2.1]

- Improvements
  - Optimized LPADC_MeasureTemperature() function to support the specific series with flash solidified calibration value.
  - Clean doxygen warnings.
- Bug Fixes
  - Fixed violations of MISRA C-2012 rule 10.3, rule 10.8 and rule 17.7.

[2.2.0]

- New Feature
  - Added API LPADC_MeasureTemperature() to get correct temperature from the internal sensor.
- Improvements
  - Separated lpadc_conversion_resolution_mode_t with related feature macro.
- Bug Fixes
  - Fixed the violations of MISRA C-2012 rules:
    - Rule 10.3, 10.4, 10.6, 10.7 and 17.7.

[2.1.1]

- Improvements
  - Updated the gain calibration formula.
  - Used feature to segregate the new item kLPADC_TriggerPriorityPreemptSubsequently.

[2.1.0]

- New Features
  - Added the API LPADC_SetOffsetValue() to support configure offset trim value manually.
  - Added the API LPADC_DoOffsetCalibration() to do offset calibration independently.
- Improvements
  - Improved the usage of macros and removed invalid macros.

[2.0.2]

- Improvements
  - Added support for platforms with 2 FIFOs and different calibration measures.

[2.0.1]

- Bug Fixes
  - Ensured the API LPADC_SetConvCommandConfig configure related registers correctly.

[2.0.0]

- Initial version.

LPCMP

[2.3.2]

- Improvements
  - Fixed LPCMP CERT-C issues.

[2.3.1]

- Improvements
  - Update LPCMP driver to be compatible with platforms that do not support LPCMP nano power mode selection.

[2.3.0]

- New Feature
  - Added some new features for platforms which support
    - Plus input source selection.
    - Minus input source selection.
    - CMP to DAC link.
- Improvements
  - Removed some new features for platforms which doesn't support
    - Functional clock source selection.
    - DAC high power mode selection.
    - Round Robin clock source selection.
    - Round Robin trigger source selection.
    - Round Robin channel sample numbers setting.
    - Round Robin channel sample time threshold setting.
    - Round Robin internal trigger configuration.

[2.2.0]

- Improvements
  - Change FSL_FEATURE_LPCMP_HAS_NO_CCR0_CMP_STOP_EN to FSL_FEATURE_LPCMP_HAS_CCR0_CMP_STOP_EN.

[2.1.3]

- New Feature
  - Added new macro to handle the case where some instances do not have the CCR0 CMP_STOP_EN bit field.

[2.1.2]

- New Feature
  - Add macros to be compatible with some platforms that do not have the CCR0 CMP_STOP_EN bitfield.

[2.1.1]

- Improvements
  - Release peripheral from reset if necessary in init function.

[2.1.0]

- New Features:
  - Supported round robin mode and window mode feature.

[2.0.3]

- Bug Fixes:
  - Fixed the violation of MISRA-2012 rule 17.7.

[2.0.2]

- Bug Fixes:
  - The current API LPCMP_ClearStatusFlags has to check w1c bits.

[2.0.1]

- Added control macro to enable/disable the CLOCK code in current driver.

[2.0.0]

- Initial version.

LPI2C

[2.6.3]

- Bug Fixes
  - Fixed static analysis identified issues.

[2.6.2]

- Improvements
  - Added timeout for while loop in LPI2C_TransferStateMachineSendCommand().

[2.6.1]

- Bug Fixes
  - Fixed coverity issues.

[2.6.0]

- New Feature
  - Added common IRQ handler entry LPI2C_DriverIRQHandler.

[2.5.7]

- Improvements
  - Added support for separated IRQ handlers.

[2.5.6]

- Improvements
  - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.5.5]

- Bug Fixes
  - Fixed LPI2C_SlaveInit() - allow to disable SDA/SCL glitch filter.

[2.5.4]

- Bug Fixes
  - Fixed LPI2C_MasterTransferBlocking() - the return value was sometime affected by call of LPI2C_MasterStop().

[2.5.3]

- Improvements
  - Added handler for LPI2C7 and LPI2C8.

[2.5.2]

- Bug Fixes
  - Fixed ERR051119 to ignore the nak flag when IGNACK=1 in LPI2C_MasterCheckAndClearError.

[2.5.1]

- Bug Fixes
  - Added bus stop incase of bus stall in LPI2C_MasterTransferBlocking.
- Improvements
  - Release peripheral from reset if necessary in init function.

[2.5.0]

- New Features
  - Added new function LPI2C_SlaveEnableAckStall to enable or disable ACKSTALL.

[2.4.1]

- Improvements
  - Before master transfer with transactional APIs, enable master function while disable slave function and vise versa for slave transfer to avoid the one affecting the other.

[2.4.0]

- Improvements
  - Split some functions, fixed CCM problem in file fsl_lpi2c.c.
- Bug Fixes
  - Fixed bug in LPI2C_MasterInit that the MCFGR2's value set in LPI2C_MasterSetBaudRate may be overwritten by mistake.

[2.3.2]

- Improvements
  - Initialized the EDMA configuration structure in the LPI2C EDMA driver.

[2.3.1]

- Improvements
  - Updated LPI2C_GetCyclesForWidth to add the parameter of minimum cycle, because for master SDA/SCL filter, master bus idle/pin low timeout and slave SDA/SCL filter configuration, 0 means disabling the feature and cannot be used.
- Bug Fixes
  - Fixed bug in LPI2C_SlaveTransferHandleIRQ that when restart detect event happens the transfer structure should not be cleared.
  - Fixed bug in LPI2C_RunTransferStateMachine, that when only slave address is transferred or there is still data remaining in tx FIFO the last byte's nack cannot be ignored.
  - Fixed bug in slave filter doze enable, that when FILTDZ is set it means disable rather than enable.
  - Fixed bug in the usage of LPI2C_GetCyclesForWidth. First its return value cannot be used directly to configure the slave FILTSDA, FILTSCL, DATAVD or CLKHOLD, because the real cycle width for them should be FILTSDA+3, FILTSCL+3, FILTSCL+DATAVD+3 and CLKHOLD+3. Second when cycle period is not affected by the prescaler value, prescaler value should be passed as 0 rather than 1.
  - Fixed wrong default setting for LPI2C slave. If enabling the slave tx SCL stall, then the default clock hold time should be set to 250ns according to I2C spec for 100kHz standard mode baudrate.
  - Fixed bug that before pushing command to the tx FIFO the FIFO occupation should be checked first in case FIFO overflow.

[2.3.0]

- New Features
  - Supported reading more than 256 bytes of data in one transfer as master.
  - Added API LPI2C_GetInstance.
- Bug Fixes
  - Fixed bug in LPI2C_MasterTransferAbortEDMA, LPI2C_MasterTransferAbort and LPI2C_MasterTransferHandleIRQ that before sending stop signal whether master is active and whether stop signal has been sent should be checked, to make sure no FIFO error or bus error will be caused.
  - Fixed bug in LPI2C master EDMA transactional layer that the bus error cannot be caught and returned by user callback, by monitoring bus error events in interrupt handler.
  - Fixed bug in LPI2C_GetCyclesForWidth that the parameter used to calculate clock cycle should be 2^prescaler rather than prescaler.
  - Fixed bug in LPI2C_MasterInit that timeout value should be configured after baudrate, since the timeout calculation needs prescaler as parameter which is changed during baudrate configuration.
  - Fixed bug in LPI2C_MasterTransferHandleIRQ and LPI2C_RunTransferStateMachine that when master writes with no stop signal, need to first make sure no data remains in the tx FIFO before finishes the transfer.

[2.2.0]

- Bug Fixes
  - Fixed issue that the SCL high time, start hold time and stop setup time do not meet I2C specification, by changing the configuration of data valid delay, setup hold delay, clock high and low parameters.
  - MISRA C-2012 issue fixed.
    - Fixed rule 8.4, 13.5, 17.7, 20.8.

[2.1.12]

- Bug Fixes
  - Fixed MISRA advisory 15.5 issues.

[2.1.11]

- Bug Fixes
  - Fixed the bug that, during master non-blocking transfer, after the last byte is sent/received, the kLPI2C_MasterNackDetectFlag is expected, so master should not check and clear kLPI2C_MasterNackDetectFlag when remainingBytes is zero, in case FIFO is emptied when stop command has not been sent yet.
  - Fixed the bug that, during non-blocking transfer slave may nack master while master is busy filling tx FIFO, and NDF may not be handled properly.

[2.1.10]

- Bug Fixes
  - MISRA C-2012 issue fixed.
    - Fixed rule 10.3, 14.4, 15.5.
  - Fixed unaligned access issue in LPI2C_RunTransferStateMachine.
  - Fixed uninitialized variable issue in LPI2C_MasterTransferHandleIRQ.
  - Used linked TCD to disable tx and enable rx in read operation to fix the issue that for platform sharing the same DMA request with tx and rx, during LPI2C read operation if interrupt with higher priority happened exactly after command was sent and before tx disabled, potentially both tx and rx could trigger dma and cause trouble.
  - Fixed MISRA issues.
    - Fixed rules 10.1, 10.3, 10.4, 11.6, 11.9, 14.4, 17.7.
  - Fixed the waitTimes variable not re-assignment issue for each byte read.
- New Features
  - Added the IRQHandler for LPI2C5 and LPI2C6 instances.
- Improvements
  - Updated the LPI2C_WAIT_TIMEOUT macro to unified name I2C_RETRY_TIMES.

[2.1.9]

- Bug Fixes
  - Fixed Coverity issue of unchecked return value in I2C_RTOS_Transfer.
  - Fixed Coverity issue of operands did not affect the result in LPI2C_SlaveReceive and LPI2C_SlaveSend.
  - Removed STOP signal wait when NAK detected.
  - Cleared slave repeat start flag before transmission started in LPI2C_SlaveSend/LPI2C_SlaveReceive. The issue was that LPI2C_SlaveSend/LPI2C_SlaveReceive did not handle with the reserved repeat start flag. This caused the next slave to send a break, and the master was always in the receive data status, but could not receive data.

[2.1.8]

- Bug Fixes
  - Fixed the transfer issue with LPI2C_MasterTransferNonBlocking, kLPI2C_TransferNoStopFlag, with the wait transfer done through callback in a way of not doing a blocking transfer.
  - Fixed the issue that STOP signal did not appear in the bus when NAK event occurred.

[2.1.7]

- Bug Fixes
  - Cleared the stopflag before transmission started in LPI2C_SlaveSend/LPI2C_SlaveReceive. The issue was that LPI2C_SlaveSend/LPI2C_SlaveReceive did not handle with the reserved stop flag and caused the next slave to send a break, and the master always stayed in the receive data status but could not receive data.

[2.1.6]

- Bug Fixes
  - Fixed driver MISRA build error and C++ build error in LPI2C_MasterSend and LPI2C_SlaveSend.
  - Reset FIFO in LPI2C Master Transfer functions to avoid any byte still remaining in FIFO during last transfer.
  - Fixed the issue that LPI2C_MasterStop did not return the correct NAK status in the bus for second transfer to the non-existing slave address.

[2.1.5]

- Bug Fixes
  - Extended the Driver IRQ handler to support LPI2C4.
  - Changed to use ARRAY_SIZE(kLpi2cBases) instead of FEATURE COUNT to decide the array size for handle pointer array.

[2.1.4]

- Bug Fixes
  - Fixed the LPI2C_MasterTransferEDMA receive issue when LPI2C shared same request source with TX/RX DMA request. Previously, the API used scatter-gather method, which handled the command transfer first, then the linked TCD which was pre-set with the receive data transfer. The issue was that the TX DMA request and the RX DMA request were both enabled, so when the DMA finished the first command TCD transfer and handled the receive data TCD, the TX DMA request still happened due to empty TX FIFO. The result was that the RX DMA transfer would start without waiting on the expected RX DMA request.
  - Fixed the issue by enabling IntMajor interrupt for the command TCD and checking if there was a linked TCD to disable the TX DMA request in LPI2C_MasterEDMACallback API.

[2.1.3]

- Improvements
  - Added LPI2C_WATI_TIMEOUT macro to allow the user to specify the timeout times for waiting flags in functional API and blocking transfer API.
  - Added LPI2C_MasterTransferBlocking API.

[2.1.2]

- Bug Fixes
  - In LPI2C_SlaveTransferHandleIRQ, reset the slave status to idle when stop flag was detected.

[2.1.1]

- Bug Fixes

  - Disabled the auto-stop feature in eDMA driver. Previously, the auto-stop feature was enabled at transfer when transferring with stop flag. Since transfer was without stop flag and the auto-stop feature was enabled, when starting a new transfer with stop flag, the stop flag would be sent before the new transfer started, causing unsuccesful sending of the start flag, so the transfer could not start.
  - Changed default slave configuration with address stall false.

[2.1.0]

- Improvements
  - API name changed:
    * LPI2C_MasterTransferCreateHandle -> LPI2C_MasterCreateHandle.
    * LPI2C_MasterTransferGetCount -> LPI2C_MasterGetTransferCount.
    * LPI2C_MasterTransferAbort -> LPI2C_MasterAbortTransfer.
    * LPI2C_MasterTransferHandleIRQ -> LPI2C_MasterHandleInterrupt.
    * LPI2C_SlaveTransferCreateHandle -> LPI2C_SlaveCreateHandle.
    * LPI2C_SlaveTransferGetCount -> LPI2C_SlaveGetTransferCount.
    * LPI2C_SlaveTransferAbort -> LPI2C_SlaveAbortTransfer.
    * LPI2C_SlaveTransferHandleIRQ -> LPI2C_SlaveHandleInterrupt.

[2.0.0]

- Initial version.

LPI2C_EDMA

[2.4.6]

- Bug Fixes
  - Fixed static analysis identified issues.

[2.4.5]

- Improvements
  - Added condition to IRQ handler to check whether the interrupt is enabled - kLPI2C_MasterTxReadyFlag.

[2.4.4]

- Improvements
  - Added support for 2KB data transfer

[2.4.3]

- Improvements
  - Added support for separated IRQ handlers.

[2.4.2]

- Improvements
  - Add EDMA ext API to accommodate more types of EDMA.

[2.4.1]

- Refer LPI2C driver change log 2.0.0 to 2.4.1

LPIT

[2.1.3]

- Bug Fixes
  - Fixed doxygen generation warnings.

[2.1.2]

- Bug Fixes
  - Fix CERT INT31-C issues.

[2.1.1]

- Improvements
  - Release peripheral from reset if necessary in init function.

[2.1.0]

- Improvements
  - Add new function LPIT_SetTimerValue to set timeout period.

[2.0.2]

- Improvements
  - Improved LPIT_SetTimerPeriod implementation, configure timeout value with LPIT ticks minus 1 generate more correct interval.
  - Added timeout value configuration check for LPIT_SetTimerPeriod, at least input 3 ticks for calling LPIT_SetTimerPeriod.
- Bug Fixes
  - Fixed MISRA C-2012 rule 17.7 violations.

[2.0.1]

- Bug Fixes
  - MISRA C-2012 issue fixed.
    - Fixed rules, containing: rule-10.3, rule-14.4, rule-15.5.

[2.0.0]

- Initial version.

LPSPI

[2.7.4]

- Bug Fixes
  - Clear WIDTH bits from the TCR register before writing a new value in LPSPI_MasterTransferBlocking().

[2.7.3]

- Improvements
  - Added timeout for while loop in LPSPI_MasterTransferWriteAllTxData().
  - Make SPI_RETRY_TIMES configurable by CONFIG_SPI_RETRY_TIMES.

[2.7.2]

- Bug Fixes
  - Fixed coverity issues.

[2.7.1]

- Bug Fixes
  - Workaround for errata ERR050607
  - Workaround for errata ERR010655

[2.7.0]

- New Feature
  - Added common IRQ handler entry LPSPI_DriverIRQHandler.

[2.6.10]

- Improvements
  - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.6.9]

- Bug Fixes
  - Fixed reading of TCR register
  - Workaround for errata ERR050606

[2.6.8]

- Bug Fixes
  - Fixed build error when SPI_RETRY_TIMES is defined to non-zero value.

[2.6.7]

- Bug Fixes
  - Fixed the txData from void * to const void * in transmit API _lpspi_master_handle and _lpspi_slave_handle.

[2.6.6]

- Bug Fixes
  - Added LPSPI register init in LPSPI_MasterInit incase of LPSPI register exist.

[2.6.5]

- Improvements
  - Introduced FSL_FEATURE_LPSPI_HAS_NO_PCSCFG and FSL_FEATURE_LPSPI_HAS_NO_MULTI_WIDTH for conditional compile.
  - Release peripheral from reset if necessary in init function.

[2.6.4]

- Bug Fixes
  - Added LPSPI6_DriverIRQHandler for LPSPI6 instance.

[2.6.3]

- Hot Fixes
  - Added macro switch in function LPSPI_Enable about ERRATA051472.

[2.6.2]

- Bug Fixes
  - Disabled lpspi before LPSPI_MasterSetBaudRate incase of LPSPI opened.

[2.6.1]

- Bug Fixes
  - Fixed return value while calling LPSPI_WaitTxFifoEmpty in function LPSPI_MasterTransferNonBlocking.

[2.6.0]

- Feature
  - Added the new feature of multi-IO SPI .

[2.5.3]

- Bug Fixes
  - Fixed 3-wire txmask of handle vaule reentrant issue.

[2.5.2]

- Bug Fixes
  - Workaround for errata ERR051588 by clearing FIFO after transmit underrun occurs.

[2.5.1]

- Bug Fixes
  - Workaround for errata ERR050456 by resetting the entire module using LPSPIn_CR[RST] bit.

[2.5.0]

- Bug Fixes
  - Workaround for errata ERR011097 to wait the TX FIFO to go empty when writing TCR register and TCR[TXMSK] value is 1.
  - Added API LPSPI_WaitTxFifoEmpty for wait the txfifo to go empty.

[2.4.7]

- Bug Fixes
  - Fixed bug that the SR[REF] would assert if software disabled or enabled the LPSPI module in LPSPI_Enable.

[2.4.6]

- Improvements
  - Moved the configuration of registers for the 3-wire lpspi mode to the LPSPI_MasterInit and LPSPI_SlaveInit function.

[2.4.5]

- Improvements
  - Improved LPSPI_MasterTransferBlocking send performance when frame size is 1-byte.

[2.4.4]

- Bug Fixes
  - Fixed LPSPI_MasterGetDefaultConfig incorrect default inter-transfer delay calculation.

[2.4.3]

- Bug Fixes
  - Fixed bug that the ISR response speed is too slow on some platforms, resulting in the first transmission of overflow, Set proper RX watermarks to reduce the ISR response times.

[2.4.2]

- Bug Fixes
  - Fixed bug that LPSPI_MasterTransferBlocking will modify the parameter txbuff and rxbuff pointer.

[2.4.1]

- Bug Fixes
  - Fixed bug that LPSPI_SlaveTransferNonBlocking can't detect RX error.

[2.4.0]

- Improvements
  - Split some functions, fixed CCM problem in file fsl_lpspi.c.

[2.3.1]

- Improvements
  - Initialized the EDMA configuration structure in the LPSPI EDMA driver.
- Bug Fixes
  - Fixed bug that function LPSPI_MasterTransferBlocking should return after the transfer complete flag is set to make sure the PCS is re-asserted.

[2.3.0]

- New Features
  - Supported the master configuration of sampling the input data using a delayed clock to improve slave setup time.

[2.2.1]

- Bug Fixes
  - Fixed bug in LPSPI_SetPCSContinous when disabling PCS continous mode.

[2.2.0]

- Bug Fixes
  - Fixed bug in 3-wire polling and interrupt transfer that the received data is not correct and the PCS continous mode is not working.

[2.1.0]

- Improvements
  - Improved LPSPI_SlaveTransferHandleIRQ to fill up TX FIFO instead of write one data to TX register which improves the slave transmit performance.
  - Added new functional APIs LPSPI_SelectTransferPCS and LPSPI_SetPCSContinous to support changing PCS selection and PCS continous mode.
- Bug Fixes

  - Fixed bug in non-blocking and EDMA transfer APIs that kStatus_InvalidArgument is returned if user configures 3-wire mode and full-duplex transfer at the same time, but transfer state is already set to kLPSPI_Busy by mistake causing following transfer can not start.
  - Fixed bug when LPSPI slave using EDMA way to transfer, tx should be masked when tx data is null, otherwise in 3-wire mode which tx/rx use the same pin, the received data will be interfered.

[2.0.5]

- Improvements
  - Added timeout mechanism when waiting certain states in transfer driver.
- Bug Fixes
  - Fixed the bug that LPSPI can not transfer large data using EDMA.
  - Fixed MISRA 17.7 issues.
  - Fixed variable overflow issue introduced by MISRA fix.
  - Fixed issue that rxFifoMaxBytes should be calculated according to transfer width rather than FIFO width.
  - Fixed issue that completion flag was not cleared after transfer completed.

[2.0.4]

- Bug Fixes
  - Fixed in LPSPI_MasterTransferBlocking that master rxfifo may overflow in stall condition.
  - Eliminated IAR Pa082 warnings.
  - Fixed MISRA issues.
    - Fixed rules 10.1, 10.3, 10.4, 10.6, 11.9, 14.2, 14.4, 15.7, 17.7.

[2.0.3]

- Bug Fixes
  - Removed LPSPI_Reset from LPSPI_MasterInit and LPSPI_SlaveInit, because this API may glitch the slave select line. If needed, call this function manually.

[2.0.2]

- New Features
  - Added dummy data set up API to allow users to configure the dummy data to be transferred.
  - Enabled the 3-wire mode, SIN and SOUT pins can be configured as input/output pin.

[2.0.1]

- Bug Fixes
  - Fixed the bug that the clock source should be divided by the PRESCALE setting in LPSPI_MasterSetDelayTimes function.

  - Fixed the bug that LPSPI_MasterTransferBlocking function would hang in some corner cases.
- Optimization
  - Added #ifndef/#endif to allow user to change the default TX value at compile time.

[2.0.0]

- Initial version.

LPSPI_EDMA

[2.4.9]

- Improvements
  - Removed unused code from LPSPI_SeparateEdmaReadData().

[2.4.8]

- Improvements
  - Added timeout for while loop in EDMA_LpspiMasterCallback() and EDMA_LpspiSlaveCallback().

[2.4.7]

- Bug Fixes
  - Add macro LPSPI_ALIGN_TCD_SIZE_MASK to align an address to edma_tcd_t size.

[2.4.6]

- Improvements
  - Increased transmit FIFO watermark to ensure whole transmit FIFO will be used during data transfer.

[2.4.5]

- Bug Fixes
  - Fixed reading of TCR register
  - Workaround for errata ERR050606

[2.4.4]

- Improvements
  - Add EDMA ext API to accommodate more types of EDMA.

[2.4.3]

- Improvements
  - Supported 32K bytes transmit in DMA, improve the max datasize in LPSPI_MasterTransferEDMALite.

[2.4.2]

- Improvements
  - Added callback status in EDMA_LpspiMasterCallback and EDMA_LpspiSlaveCallback to check transferDone.

[2.4.1]

- Improvements
  - Add the TXMSK wait after TCR setting.

[2.4.0]

- Improvements
  - Separated LPSPI_MasterTransferEDMA functions to LPSPI_MasterTransferPrepareEDMA and LPSPI_MasterTransferEDMALite to optimize the process of transfer.

LPTMR

[2.2.1]

- Bug Fixes
  - Fix CERT INT31-C issues.

[2.2.0]

- Improvements
  - Updated lptmr_prescaler_clock_select_t, only define the valid options.

[2.1.1]

- Improvements
  - Updated the characters from "PTMR" to "LPTMR" in "FSL_FEATURE_PTMR_HAS_NO_PRESCALER_CLOCK_SOURCE_1_SUPPORT" feature definition.

[2.1.0]

- Improvements
  - Implement for some special devices' not supporting for all clock sources.
- Bug Fixes
  - Fixed issue when accessing CMR register.

[2.0.2]

- Bug Fixes
  - Fixed MISRA-2012 issues.
    - Rule 10.1.

[2.0.1]

- Improvements
  - Updated the LPTMR driver to support 32-bit CNR and CMR registers in some devices.

[2.0.0]

- Initial version.

LPUART

[2.10.0]

- New Feature
  - Added support to configure RTS watermark.

[2.9.4]

- Improvements
  - Merged duplicate code.

[2.9.3]

- Improvements
  - Added timeout for while loops in LPUART_Deinit().

[2.9.2]

- Bug Fixes
  - Fixed coverity issues.

[2.9.1]

- Bug Fixes
  - Fixed coverity issues.

[2.9.0]

- New Feature
  - Added support for swap TXD and RXD pins.
  - Added common IRQ handler entry LPUART_DriverIRQHandler.

[2.8.3]

- Improvements
  - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.8.2]

- Bug Fix
  - Fixed the bug that LPUART_TransferEnable16Bit controled by wrong feature macro.

[2.8.1]

- Bug Fixes
  - Fixed issue for MISRA-2012 check.
    - Fixed rule-5.3, rule-5.8, rule-10.4, rule-11.3, rule-11.8.

[2.8.0]

- Improvements
  - Added support of DATA register for 9bit or 10bit data transmit in write and read API. Such as: LPUART_WriteBlocking16bit, LPUART_ReadBlocking16bit, LPUART_TransferEnable16Bit LPUART_WriteNonBlocking16bit, LPUART_ReadNonBlocking16bit.

[2.7.7]

- Bug Fixes
  - Fixed the bug that baud rate calculation overflow when srcClock_Hz is 528MHz.

[2.7.6]

- Bug Fixes
  - Fixed LPUART_EnableInterrupts and LPUART_DisableInterrupts bug that blocks if the LPUART address doesn't support exclusive access.

[2.7.5]

- Improvements
  - Release peripheral from reset if necessary in init function.

[2.7.4]

- Improvements
  - Added support for atomic register accessing in LPUART_EnableInterrupts and LPUART_DisableInterrupts.

[2.7.3]

- Bug Fixes
  - Fixed violations of the MISRA C-2012 rules 15.7.

[2.7.2]

- Bug Fix
  - Fixed the bug that the OSR calculation error when lupart init and lpuart set baud rate.

[2.7.1]

- Improvements
  - Added support for LPUART_BASE_PTRS_NS in security mode in file fsl_lpuart.c.

[2.7.0]

- Improvements
  - Split some functions, fixed CCM problem in file fsl_lpuart.c.

[2.6.0]

- Bug Fixes
  - Fixed bug that when there are multiple lpuart instance, unable to support different ISR.

[2.5.3]

- Bug Fixes
  - Fixed comments by replacing unused status flags kLPUART_NoiseErrorInRxDataRegFlag and kLPUART_ParityErrorInRxDataRegFlag with kLPUART_NoiseErrorFlag and kLPUART_ParityErrorFlag.

[2.5.2]

- Bug Fixes
  - Fixed bug that when setting watermark for TX or RX FIFO, the value may exceed the maximum limit.
- Improvements
  - Added check in LPUART_TransferDMAHandleIRQ and LPUART_TransferEdmaHandleIRQ to ensure if user enables any interrupts other than transfer complete interrupt, the dma transfer is not terminated by mistake.

[2.5.1]

- Improvements
  - Use separate data for TX and RX in lpuart_transfer_t.
- Bug Fixes
  - Fixed bug that when ring buffer is used, if some data is received in ring buffer first before calling LPUART_TransferReceiveNonBlocking, the received data count returned by LPUART_TransferGetReceiveCount is wrong.

[2.5.0]

- Bug Fixes
  - Added missing interrupt enable masks kLPUART_Match1InterruptEnable and kLPUART_Match2InterruptEnable.
  - Fixed bug in LPUART_EnableInterrupts, LPUART_DisableInterrupts and LPUART_GetEnabledInterrupts that the BAUD[LBKDIE] bit field should be soc specific.

  - Fixed bug in LPUART_TransferHandleIRQ that idle line interrupt should be disabled when rx data size is zero.
  - Deleted unused status flags kLPUART_NoiseErrorInRxDataRegFlag and kLPUART_ParityErrorInRxDataRegFlag, since firstly their function are the same as kLPUART_NoiseErrorFlag and kLPUART_ParityErrorFlag, secondly to obtain them one data word must be read out thus interfering with the receiving process.
  - Fixed bug in LPUART_GetStatusFlags that the STAT[LBKDIF], STAT[MA1F] and STAT[MA2F] should be soc specific.
  - Fixed bug in LPUART_ClearStatusFlags that tx/rx FIFO is reset by mistake when clearing flags.
  - Fixed bug in LPUART_TransferHandleIRQ that while clearing idle line flag the other bits should be masked in case other status bits be cleared by accident.
  - Fixed bug of race condition during LPUART transfer using transactional APIs, by disabling and re-enabling the global interrupt before and after critical operations on interrupt enable register.
  - Fixed DMA/eDMA transfer blocking issue by enabling tx idle interrupt after DMA/eDMA transmission finishes.
- New Features
  - Added APIs LPUART_GetRxFifoCount/LPUART_GetTxFifoCount to get rx/tx FIFO data count.
  - Added APIs LPUART_SetRxFifoWatermark/LPUART_SetTxFifoWatermark to set rx/tx FIFO water mark.

[2.4.1]

- Bug Fixes
  - Fixed MISRA advisory 17.7 issues.

[2.4.0]

- New Features
  - Added APIs to configure 9-bit data mode, set slave address and send address.

[2.3.1]

- Bug Fixes
  - Fixed MISRA advisory 15.5 issues.

[2.3.0]

- Improvements
  - Modified LPUART_TransferHandleIRQ so that txState will be set to idle only when all data has been sent out to bus.
  - Modified LPUART_TransferGetSendCount so that this API returns the real byte count that LPUART has sent out rather than the software buffer status.
  - Added timeout mechanism when waiting for certain states in transfer driver.

[2.2.8]

- Bug Fixes
  - Fixed issue for MISRA-2012 check.
    - Fixed rule-10.3, rule-14.4, rule-15.5.
  - Eliminated Pa082 warnings by assigning volatile variables to local variables and using local variables instead.
  - Fixed MISRA issues.
    - Fixed rules 10.1, 10.3, 10.4, 10.8, 14.4, 11.6, 17.7.
- Improvements
  - Added check for kLPUART_TransmissionCompleteFlag in LPUART_WriteBlocking, LPUART_TransferHandleIRQ, LPUART_TransferSendDMACallback and LPUART_SendEDMACallback to ensure all the data would be sent out to bus.
  - Rounded up the calculated sbr value in LPUART_SetBaudRate and LPUART_Init to achieve more acurate baudrate setting. Changed osr from uint32_t to uint8_t since osr's bigest value is 31.
  - Modified LPUART_ReadBlocking so that if more than one receiver errors occur, all status flags will be cleared and the most severe error status will be returned.

[2.2.7]

- Bug Fixes
  - Fixed issue for MISRA-2012 check.
    - Fixed rule-12.1, rule-17.7, rule-14.4, rule-13.3, rule-14.4, rule-10.4, rule-10.8, rule-10.3, rule-10.7, rule-10.1, rule-11.6, rule-13.5, rule-11.3, rule-13.2, rule-8.3.

[2.2.6]

- Bug Fixes
  - Fixed the issue of register's being in repeated reading status while dealing with the IRQ routine.

[2.2.5]

- Bug Fixes
  - Do not set or clear the TIE/RIE bits when using LPUART_EnableTxDMA and LPUART_EnableRxDMA.

[2.2.4]

- Improvements
  - Added hardware flow control function support.
  - Added idle-line-detecting feature in LPUART_TransferNonBlocking function. If an idle line is detected, a callback is triggered with status kStatus_LPUART_IdleLineDetected returned. This feature may be useful when the received Bytes is less than the expected received data size. Before triggering the callback, data in the FIFO (if has FIFO) is read out, and no interrupt will be disabled, except for that the receive data size reaches 0.

  - Enabled the RX FIFO watermark function. With the idle-line-detecting feature enabled, users can set the watermark value to whatever you want (should be less than the RX FIFO size). Data is received and a callback will be triggered when data receive ends.

[2.2.3]

- Improvements
  - Changed parameter type in LPUART_RTOS_Init struct from rtos_lpuart_config to lpuart_rtos_config_t.
- Bug Fixes
  - Disabled LPUART receive interrupt instead of all NVICs when reading data from ring buffer. Otherwise when the ring buffer is used, receive nonblocking method will disable all NVICs to protect the ring buffer. This may has a negative effect on other IPs that are using the interrupt.

[2.2.2]

- Improvements
  - Added software reset feature support.
  - Added software reset API in LPUART_Init.

[2.2.1]

- Improvements
  - Added separate RX/TX IRQ number support.

[2.2.0]

- Improvements
  - Added support of 7 data bits and MSB.

[2.1.1]

- Improvements
  - Removed unnecessary check of event flags and assert in LPUART_RTOS_Receive.
  - Added code to always wait for RX event flag in LPUART_RTOS_Receive.

[2.1.0]

- Improvements
  - Update transactional APIs.

LPUART_EDMA

[2.4.0]

- Refer LPUART driver change log 2.1.0 to 2.4.0

LTC

[2.0.18]

- Bug fix:
- Fix CERT INT30-C and INT31-C compliance.

[2.0.17]

- Bug fix:
  - Fix CMAC for payloads over one block, and if BRIC is present on the device, remove XCBC and "decrypt key" functionality

[2.0.16]

- Bug fix:
  - Fix unitialized GCC warning in LTC_AES_GenerateDecryptKey().

[2.0.15]

- Bug fix:
  - Fix MISRA-2012 issues.

[2.0.14]

- Improvements:
  - Add feature macro FSL_FEATURE_LTC_HAS_NO_CLOCK_CONTROL_BIT into LTC_Deinit function.

[2.0.13]

- Improvements:
  - Add feature macro FSL_FEATURE_LTC_HAS_NO_CLOCK_CONTROL_BIT into LTC_init function.

[2.0.12]

- Bug fix:
  - Fix AES Decrypt in CBC modes fail when used kLTC_DecryptKey.

[2.0.11]

- Bug fix:
  - Fix MISRA-2012 issues.

[2.0.10]

- Bug fix:
  - Fix MISRA-2012 issues.

[2.0.9]

- Bug fix:
  - Fix sign-compare warning in ltc_set_context and in ltc_get_context.

[2.0.8]

- Bug fix:
  - Fixed coverity issues introduced in 2.0.7

[2.0.7]

- Bug fix:
  - Fixed MISRA-2012 issues
    - Fixed rule 15.7. 10.1. 8.6. 10.7. 10.3, 11.6, 11.3, 10.8, 16.1.

[2.0.6]

- Bug fix:
  - Fixed [KPSDK-23603][LTC] AES Decrypt in ECB and CBC modes fail when ciphertext size > 0xff0 bytes.

[2.0.5]

- Improvements:
  - Fixed MISRA issues
    - Fixed rule 15.7, rule 14.4, rule 10.4, rule 18.4, rule 17.7, 5.7, 12.2.

[2.0.4]

- Improvements:
  - Constant LTC_PKHA_CompareBigNum() processing time.

[2.0.3]

- Bug fix:
  - Fixed LTC_PKHA_CompareBigNum() in case an integer argument is an array of all zeros.

[2.0.2]

- Bug fix:
  - Fixed [KPSDK-10932][LTC][SHA] LTC_HASH() blocks indefinitely when message size exceeds 4080 Bytes.

[2.0.1]

- Bug fix:
  - Fixed c++ build warning in block_to_ififo() functions.

[2.0.0]

- Initial version.

MCM

[2.2.0]

- Improvements
  - Support platforms with less features.

[2.1.0]

- Others
  - Remove byteID from mcm_lmem_fault_attribute_t for document update.

[2.0.0]

- Initial version.

MSCM

[2.0.0]

- Initial version.

PORT

[2.5.1]

- Bug Fixes
  - Fix CERT INT31-C issues.
  - Fixed the violations of MISRA C-2012 rules: 10.1.

[2.5.0]

- Bug Fixes
  - Correct the kPORT_MuxAsGpio for some platforms.

[2.4.1]

- Bug Fixes
  - Fixed the violations of MISRA C-2012 rules: 10.1, 10.8 and 14.4.

[2.4.0]

- New Features
  - Updated port_pin_config_t to support input buffer and input invert.

[2.3.0]

- New Features
  - Added new APIs for Electrical Fast Transient(EFT) detect.
  - Added new API to configure port voltage range.

[2.2.0]

- New Features
  - Added new api PORT_EnablePinDoubleDriveStrength.

[2.1.1]

- Bug Fixes
  - Fixed the violations of MISRA C-2012 rules: 10.1, 10.4□11.3□11.8, 14.4.

[2.1.0]

- New Features
  - Updated the driver code to adapt the case of the interrupt configurations in GPIO module. Will move the pin configuration APIs to GPIO module.

[2.0.2]

- Other Changes
  - Added feature guard macros in the driver.

[2.0.1]

- Other Changes
  - Added “const” in function parameter.
  - Updated some enumeration variables’ names.

RTC

[2.4.0]

- New features
  - Add support for RTC clock output.
  - Add support for RTC time seconds interrupt configuration.

[2.3.3]

- Bug Fixes
  - Fix RTC_GetDatetime function validating datetime issue.

[2.3.2]

- Improvements
  - Handle errata 010716: Disable the counter before setting alarm register and then reenable the counter.

[2.3.1]

- Bug Fixes
  - Fixed CERT INT31-C violations.

[2.3.0]

- Improvements
  - Added API RTC_EnableLPOClock to set 1kHz LPO clock.
  - Added API RTC_EnableCrystalClock to replace API RTC_SetClockSource.

[2.2.2]

- Improvements
  - Refine _rtc_interrupt_enable order.

[2.2.1]

- Bug Fixes
  - Fixed the issue of Pa082 warning.
  - Fixed the issue of bit field mask checking.
  - Fixed the issue of hard code in RTC_Init.

[2.2.0]

- Bug Fixes
  - Fixed MISRA C-2012 issue.
    - Fixed rule contain: rule-17.7, rule-14.4, rule-10.4, rule-10.7, rule-10.1, rule-10.3.
  - Fixed central repository code formatting issue.
- Improvements
  - Added an API for enabling wakeup pin.

[2.1.0]

- Improvements
  - Added feature macro check for many features.

[2.0.0]

- Initial version.

SEMA42

[2.1.1]

- Improvements
  - Updated SEMA42_TryLock function to avoid unsigned integer operations wrap issue.

[2.1.0]

- New Features
  - Added SEMA42_BUSY_POLL_COUNT parameter to prevent infinite polling loops in SEMA42 operations.
  - Added timeout mechanism to all polling loops in SEMA42 driver code.
- Improvements
  - Updated SEMA42_Lock function to return status_t instead of void for better error handling.
  - Enhanced documentation to clarify timeout behavior and return values.

[2.0.4]

- Improvements
  - Release peripheral from reset if necessary in init function.

[2.0.3]

- Improvements
  - Changed to implement SEMA42_Lock base on SEMA42_TryLock.

[2.0.2]

- Bug Fixes
  - Fixed violations of the MISRA C-2012 rules 17.7.

[2.0.1]

- Bug Fixes
  - Fixed violations of the MISRA C-2012 rules 10.3, 10.4, 14.4, 18.1.

[2.0.0]

- Initial version.

SFA

[2.1.4]

- Bug Fixes
  - Fixed doxygen warnings: unbalanced grouping commands.

[2.1.3]

- Improvements
  - Add timeout for APIs with dfmea issues.

[2.1.2]

- Improvements
  - Updated SFA_ClearStatusFlag() function to support clearing all status flags.

[2.1.1]

- Improvements
  - Updated SFA driver code to reduce code redundancy.
  - Updated the use of macros in the driver code to enable the CUT pin function.

[2.1.0]

- Improvements
  - Updated how the clock frequency under test is calculated.
  - Supported configuration reference clock source.
  - Added clock count limit APIs.
  - Added macros to support fields that are configurable in different instances.

[2.0.1]

- Bug Fixes
  - Fixed the issue in SFA_Mode0Calculate function.
- Improvements
  - For the devices that support RF SFA, cast the peripheral structure to SFA_Type.

[2.0.0]

- Initial version.

SMSCM

[2.0.0]

- Initial version.

SPC

[2.8.0]

- Improvements
  - Added feature macros to support some platforms which do not have system LDO, VDD_SYS, DCDC_BURST_CFG.
- Bug Fixes
  - Fixed doxygen warnings:
    1. unbalanced grouping commands.
    2. Argument 'base' of command @param is not found in the argument list of sfa_callback_t(status_t status)
    3. Explicit link request to 'kStatus_SPC_CORELDOVoltageSetFail.' could not be resolved

[2.7.1]

- Improvements
  - Make configuration structure pointers const in SPC driver API for Higher Power mode

[2.7.0]

- Improvements
  - Added new APIs to control VDD Core Glitch Detector.

[2.6.1]

- Improvements
  - Add timeout for while loop code.

[2.6.0]

- New Features
  - Added new feature FSL_FEATURE_SPC_HAS_SC_SPC_LP_REQ_BIT for compatibility with new platforms that do not support the SPC SC[SPC_LP_REQ] bitfield.

[2.5.0]

- New Features
  - Added new function group to support high power feature.

[2.4.1]

- Improvements
  - Added the new feature FSL_FEATURE_SPC_HAS_SC_REG_BUSY to support the device which has the SC[REG_BUSY] bitfield.

[2.4.0]

- Improvements
  - Set functions of VD_SYS_CFG[LVSEL] configuration as deprecated, since this bit filed is reserved for all devices.

[2.3.0]

- Improvements
  - Added SPC_SetSRAMOperateVoltage() to set SRAM operate voltage.
- Bug Fixes
  - Fixed the issue in SPC_SetDCDCBurstConfig() function.
  - Fixed a bug that is SPC has VDD_DS the voltage is not configured correctly.

[2.2.0]

- Bug Fixes
  - The default value of ACTIVE_CFG[CORELDO_VDD_DS] bit is 1, removed code of set that bit.
  - Fixed an issue of SPC_SetDCDCBurstConfig() to avoid program stuck in a while loop.
  - Removed some duplicate code that never reached.
- Improvements
  - Added two APIs to control Vdd core glitch detect feature in Active/lowpower modes.

[2.1.0]

- Improvements
  - Improved the spc_dcdc_voltage_level_t enumeration.
  - Added some low power request check APIs.
  - Added SPC_SoftwareGatePowerSwitch() and SPC_PowerModeControlPowerSwitch() function.
  - If APIs' parameters is in type of structure pointer, set it to const structure pointer.
- Bug Fixes
  - Fixed the issue in SPC_SetDCDCRefreshCount() function.
  - Fixed some issues in voltage detect enable APIs.
  - Fixed some issues in Bandgap control APIs.
  - Fixed some issues in regulator active/lowpower mode configuration APIs.

[2.0.2]

- Bug Fixes
  - Fixed the doxygen issue by adding missed comments.

[2.0.1]

- Bug Fixes
  - The enumerator spc_core_ldo_voltage_level_t may different for different devices, added the macro to separate different scenes.

[2.0.0]

- Initial version.

SYSPM

[2.3.1]

- Improvements
  - Add timeout for while loop.

[2.3.0]

- New Features
  - Supported instruction counter.

[2.2.0]

- New Features
  - Supported platforms which only have one monitor.
  - Supported platforms whose instrction counter can't be cleared.
  - Supported platforms whose counter can't be disabled.

[2.1.0]

- New Features
  - Added new APIs SYSPM_Init and SYSPM_Deinit.

[2.0.0]

- Initial version.

TPM

[2.4.1]

- Improvements
  - Add Coverage Justification for uncovered code.

[2.4.0]

- New Feature
  - Added while loop timeout for MOD CnV CnSC and SC register write sequence.
  - Change the return type from void to status_t for following API:
    - TPM_DisableChannel
    - TPM_EnableChannel
    - TPM_SetupOutputCompare
    - TPM_SetTimerPeriod
    - TPM_StopTimer

[2.3.6]

- Bug Fixes
  - Fixed CERT INT30-C INT31-C issue for TPM_SetupDualEdgeCapture.

[2.3.5]

- New Feature
  - Added IRQ handler entry for TPM2.

[2.3.4]

- New Feature
  - Added common IRQ handler entry TPM_DriverIRQHandler.

[2.3.3]

- Improvements
  - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.3.2]

- Bug Fixes
  - Fixed ERR008085 TPM writing the TPMx_MOD or TPMx_CnV registers more than once may fail when the timer is disabled.

[2.3.1]

- Bug Fixes
  - Fixed compilation error when macro FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL is 1.

[2.3.0]

- Improvements
  - Create callback feature for TPM match and timer overflow interrupts.

[2.2.4]

- Improvements
  - Add feature macros(FSL_FEATURE_TPM_HAS_GLOBAL_TIME_BASE_EN, FSL_FEATURE_TPM_HAS_GLOBAL_TIME_BASE_SYNC).

[2.2.3]

- Improvements
  - Release peripheral from reset if necessary in init function.

[2.2.2]

- Bug Fixes
  - Fixed violations of MISRA C-2012 rule 10.4.

[2.2.1]

- Bug Fixes
  - Fixed CCM issue by splitting function from TPM_SetupPwm() function to reduce function complexity.
  - Fixed violations of MISRA C-2012 rule 17.7.

[2.2.0]

- Improvements
  - Added TPM_SetChannelPolarity to support select channel input/output polarity.
  - Added TPM_EnableChannelExtTrigger to support enable external trigger input to be used by channel.
  - Added TPM_CalculateCounterClkDiv to help calculates the counter clock prescaler.
  - Added TPM_GetChannelValue to support get TPM channel value.
  - Added new TPM configuration.
    - syncGlobalTimeBase
    - extTriggerPolarity
    - chnlPolarity
  - Added new PWM signal configuration.
    - secPauseLevel
- Bug Fixes
  - Fixed TPM_SetupPwm can't configure 0% combined PWM issues.

[2.1.1]

- Improvements
  - Add feature macro for PWM pause level select feature.

[2.1.0]

- Improvements
  - Added TPM_EnableChannel and TPM_DisableChannel APIs.
  - Added new PWM signal configuration.
    - pauseLevel - Support select output level when counter first enabled or paused.
    - enableComplementary - Support enable/disable generate complementary PWM signal.
    - deadTimeValue - Support deadtime insertion for each pair of channels in combined PWM mode.
- Bug Fixes
  - Fixed issues about channel MSnB:MSnA and ELSnB:ELSnA bit fields and CnV register change request acknowledgement. Writes to these bits are ignored when the interval between successive writes is less than the TPM clock period.

[2.0.8]

- Bug Fixes
  - Fixed violations of MISRA C-2012 rule 10.1, 10.4 ,10.7 and 14.4.

[2.0.7]

- Bug Fixes
  - Fixed violations of MISRA C-2012 rule 10.4 and 17.7.

[2.0.6]

- Bug Fixes
  - Fixed Out-of-bounds issue.

[2.0.5]

- Bug Fixes
  - Fixed MISRA-2012 rules.
    - Rule 10.6, 10.7

[2.0.4]

- Bug Fixes
  - Fixed ERR050050 in functions TPM_SetupPwm/TPM_UpdatePwmDutycycle. When TPM was configured in EPWM mode as PS = 0, the compare event was missed on the first reload/overflow after writing 1 to the CnV register.

[2.0.3]

- Bug Fixes
  - MISRA-2012 issue fixed.
    - Fixed rules: rule-12.1, rule-17.7, rule-16.3, rule-14.4, rule-1.3, rule-10.4, rule-10.3, rule-10.7, rule-10.1, rule-10.6, and rule-18.1.

[2.0.2]

- Bug Fixes
  - Fixed issues in functions TPM_SetupPwm/TPM_UpdateChnlEdgeLevelSelect /TPM_SetupInputCapture/TPM_SetupOutputCompare/TPM_SetupDualEdgeCapture, wait acknowledgement when the channel is disabled.

[2.0.1]

- Bug Fixes
  - Fixed TPM_UpdateChnIEdgeLevelSelect ACK wait issue.
  - Fixed the issue that TPM_SetupdualEdgeCapture could not set FILTER register.
  - Fixed TPM_UpdateChnEdgeLevelSelect ACK wait issue.

[2.0.0]

- Initial version.

TRDC

[2.3.1]

- Improvements
  - Update APIs to check whether the memory access configuration can be updated.
  - Update APIs to mask the MRC address since only high 18 bits are valid.

[2.3.0]

- New Features
  - Added API TRDC_EnableProcessorDomainAssignment to disable/enable DAC.
- Bug Fixes
  - Fixed MISRA violation of missing function declaration.
  - Fixed wrong operation of domain mask in TRDC_MbcNseClearAll and TRDC_MrcDomainNseClear.

[2.2.1]

- Bug Fixes
  - Fixed MISRA violations of rule 10.3.

[2.2.0]

- New Features
  - Added new APIs to support SoC with more than one processor core.

[2.1.1]

- Bug Fixes
  - Fixed MISRA violations of rule 10.3, 10.6, 10.7 and 18.1.

[2.1.0]

- Improvements
  - Modified some of the addressing method according to the device header file's update.
  - Modified flash address configuration structure and default setting.
- New Features
  - Added API to set flash logic window array address.
- Bug Fixes
  - Fixed wrong return value of TRDC_GetFlashLogicalWindowPbase.
  - Fixed bug in TRDC_GetAndClearFirstSpecificDomainError that the error status is cleared by mistake before obtained by software.
  - Fixed MISRA violations of rule 10.3, 10.4, 10.6 and 10.8.

[2.0.0]

- Initial version.

TRGMUX

[2.0.1]

- Bug Fixes
  - Fixed violations of the MISRA C-2012 rules 10.1, 10.3, 10.8.

[2.0.0]

- Initial version.

TSTMR

[2.1.0]

- New Features
  - Support configured clock frequency.
  - Add TSTMR_Init and TSTMR_init APIs.
- Improvements
  - Change TSTMR_DelayUs from static inline function to normal function.

[2.0.4]

- Bugfix
  - Fix MISRA C-2012 Rule 10.4 and 14.4 issues.

[2.0.3]

- Bugfix
  - Fix CERT INT30-C that Unsigned integer operation TSTMR_ReadTimeStamp(base) - startTime may wrap.

[2.0.2]

- Improvements
  - Support 24MHz clock source.
- Bugfix
  - Fix MISRA C-2012 Rule 10.4 issue.
  - Read of TSTMR HIGH must follow TSTMR LOW atomically: require masking interrupt around 2 LSB / MSB accesses.

[2.0.1]

- Bugfix
  - Restrict to read with 32-bit accesses only.
  - Restrict that TSTMR LOW read occurs first, followed by the TSTMR HIGH read.

[2.0.0]

- Initial version.

VBAT

[2.1.1]

- Improvements
  - Added timeout for while loop.

[2.1.0]

- Improvements
  - Both FROCFGA and LDOCFGA registers are set as internal register, removed corresponding interfaces.

[2.0.0]

- Initial version.

VREF

[2.4.1]

- Improvements
  - Improve doxygen documentation.

[2.4.0]

- Improvements
  - Support TEST_UNLOCK and TRIM0 registers read and write.

[2.3.0]

- Improvements
  - Add feature macros for compatibility with some platforms that don't have CSR[LPBG_BUF_EN] and UTRIM[TRIM2V1].

[2.2.2]

- Bug Fixes
  - Fixed typo in vref_buffer_mode_t enumerator.

[2.2.1]

- Improvements
  - Updated VREF driver code to reduce code redundancy.

[2.2.0]

- Improvements
  - Remove API VREF_SetTrimVal and VREF_GetTrimVal
  - Add new API VREF_SetVrefTrimVal,VREF_SetTrim21Val,VREF_GetVrefTrimVal and VREF_GetTrim21Val
  - Updated VREF driver code to conform to VREF IP architecture.

[2.1.0]

- Improvements
  - Remove API VREF_SetTrimVal and VREF_GetTrimVal
  - Add new API VREF_SetVrefTrimVal,VREF_SetTrim21Val,VREF_GetVrefTrimVal and VREF_GetTrim21Val
  - Updated VREF driver code to conform to VREF IP architecture.
- Bug Fixes
  - Fix issue which progeam stuck in VREF_Init

[2.0.0]

- Initial version.

WDOG32

[2.2.1]

- Bug Fixes
  - Fix CERT INT31-C that the bool value shall be converted to unsigned int 0 or 1 then passed to registers.
  - Fix MISRA 2012 20.3 vilation.

[2.2.0]

- Improvements
  - Added while loop timeout config value for WDOG32 reconfiguration and unlock sequence.
  - Change the return type of WDOG32_Init, WDOG32_Deinit and WDOG32_Unlock from void to status_t.

[2.1.0]

- Improvements
  - Release peripheral from reset if necessary in init function.

[2.0.4]

- Improvements
  - To ensure that the reconfiguration is inside 128 bus clocks unlock window, put all reconfiguration APIs in quick access code section.

[2.0.3]

- Bug Fixes
  - Fixed the noncompliance issue of the reference document.
    - Waited until for new configuration to take effect by checking the RCS bit field.
    - Waited until for registers to be unlocked by checking the ULK bit field.
- Improvements
  - Added 128 bus clocks delay ensures a smooth transition before restarting the counter with the new configuration when there is no RCS status bit.

[2.0.2]

- Bug Fixes
  - MISRA C-2012 issue fixed.
    - Fixed rules, containing: rule-10.3, rule-14.4, rule-15.5.
  - Fixed the issue of the inseparable process interrupted by other interrupt source.
    - WDOG32_Refresh

[2.0.1]

- Bug Fixes
  - WDOG must be configured within its configuration time period.
    - Added WDOG32_Init API to quick access section.
    - Defined register variable in WDOG32_Init API.

[2.0.0]

- Initial version.

WUU

[2.4.1]

- Improvements
  - The semantics of kWUU_FilterActiveDSPD and kWUU_ExternalPinActiveDSPD are not clear enough, because on some platforms it's not 'DSPD' (deep sleep, power down) but PDDPD (power down, deep power down). We deprecate them and will use the more generic kWUU_FilterActiveLowLeakage and kWUU_ExternalPinActiveLowLeakage in the future.

[2.4.0]

- New Features
  - Added WUU_ClearExternalWakeupPinsConfig() to clear settings of PDC and PE register.

[2.3.0]

- New Features
  - Added WUU_ClearInternalWakeUpModulesConfig() to clear settings of DM and ME register.

[2.2.1]

- Bug Fixes
  - Fixed WUU_SetPinFilterConfig() unable to set edge detection of pin filter config.
  - Fixed wrong macro used in WUU_GetPinFilterFlag() function.

[2.2.0]

- New Features
  - Added the WUU_GetExternalWakeupPinFlag() and WUU_ClearExternalWakeupPinFlag() function .

[2.1.0]

- New Features
  - Added the WUU_GetModuleInterruptFlag() function to support the devices that equipped MF register.

[2.0.0]

- Initial version.

1.6 Driver API Reference Manual

This section provides a link to the Driver API RM, detailing available drivers and their usage to help you integrate hardware efficiently.

KW45B41Z83

1.7 Middleware Documentation

Find links to detailed middleware documentation for key components. While not all onboard middleware is covered, this serves as a useful reference for configuration and development.

1.7.1 Wireless Bluetooth LE host stack and applications

examples__wireless_examples__bluetooth_docs

1.7.2 Wireless zigbee stack

Wireless Zigbee

1.7.3 Wireless Connectivity Framework

framework

1.7.4 Multicore

Multicore SDK

1.7.5 FreeMASTER

freemaster

1.7.6 FreeRTOS

FreeRTOS

Chapter 2

KW45B41Z83

2.1 CACHE: LPCAC CACHE Memory Controller

static inline void L1CACHE_EnableCodeCache(void)

Enables the processor code bus cache.

static inline void L1CACHE_DisableCodeCache(void)

Disables the processor code bus cache.

static inline void L1CACHE_InvalidateCodeCache(void)

Invalidates the processor code bus cache.

void L1CACHE_InvalidateICacheByRange(uint32_t address, uint32_t size_byte)

Invalidates L1 instrument cache by range.

Parameters

- address – The start address of the memory to be invalidated.
- size_byte – The memory size.

static inline void L1CACHE_InvalidateDCacheByRange(uint32_t address, uint32_t size_byte)

Invalidates L1 data cache by range.

Parameters

- address – The start address of the memory to be invalidated.
- size_byte – The memory size.

static inline void L1CACHE_CleanDCacheByRange(uint32_t address, uint32_t size_byte)

Cleans L1 data cache by range.

The cache is write through mode, so there is nothing to do with the cache flush/clean operation.

Parameters

- address – The start address of the memory to be cleaned.
- size_byte – The memory size.

static inline void L1CACHE_CleanInvalidateDCacheByRange(uint32_t address, uint32_t size_byte)

Cleans and Invalidates L1 data cache by range.

Parameters

- address – The start address of the memory to be clean and invalidated.

- size_byte – The memory size.

static inline void ICACHE_InvalidateByRange(uint32_t address, uint32_t size_byte)
Invalidates instruction cache by range.

Parameters

- address – The physical address.
- size_byte – size of the memory to be invalidated.

static inline void DCACHE_InvalidateByRange(uint32_t address, uint32_t size_byte)
Invalidates data cache by range.

Parameters

- address – The physical address.
- size_byte – size of the memory to be invalidated.

static inline void DCACHE_CleanByRange(uint32_t address, uint32_t size_byte)
Clean data cache by range.

Parameters

- address – The physical address.
- size_byte – size of the memory to be cleaned.

static inline void DCACHE_CleanInvalidateByRange(uint32_t address, uint32_t size_byte)
Cleans and Invalidates data cache by range.

Parameters

- address – The physical address.
- size_byte – size of the memory to be Cleaned and Invalidated.

FSL_CACHE_DRIVER_VERSION
cache driver version 2.1.2.

2.2 CCM32K: 32kHz Clock Control Module

void CCM32K_Enable32kFro(CCM32K_Type *base, bool enable)
Enable/Disable 32kHz free-running oscillator.

Note: There is a start up time before clocks are output from the FRO.

Note: To enable FRO32k and set it as 32kHz clock source please follow steps:

```
CCM32K_Enable32kFro(base, true);        //Enable FRO analog oscillator.
CCM32K_DisableCLKOutToPeripherals(base, mask); //Disable clock out.
CCM32K_SelectClockSource(base, kCCM32K_ClockSourceSelectFro32k); //Select FRO32k as clock␣
↪source.
while(CCM32K_GetStatus(base) != kCCM32K_32kFroActiveStatusFlag); //Check FOR32k is active␣
↪and in used.
CCM32K_EnableCLKOutToPeripherals(base, mask); //Enable clock out if needed.
```

Parameters

- base – CCM32K peripheral base address.

- enable – Boolean value to enable or disable the 32kHz free-running oscillator. true — Enable 32kHz free-running oscillator. false — Disable 32kHz free-running oscillator.

static inline void CCM32K_Lock32kFroWriteAccess(CCM32K_Type *base)

Lock all further write accesses to the FRO32K_CTRL register until a POR occurs.

Parameters

- base – CCM32K peripheral base address.

static inline uint16_t CCM32K_Get32kFroTrimValue(CCM32K_Type *base)

Get frequency trim value of 32kHz free-running oscillator.

Parameters

- base – CCM32K peripheral base address.

Returns

The current trim value.

void CCM32K_Set32kFroTrimValue(CCM32K_Type *base, uint16_t trimValue)

Set the frequency trim value of 32kHz free-running oscillator by software.

Note: The frequency is decreased monotonically when the trimValue is changed progressively from 0x0U to 0x7FFU.

Note: If the FRO32 is enabled before invoking this function, then in this function the FRO32 will be disabled, after updating trim value the FRO32 will be re-enabled.

Parameters

- base – CCM32K peripheral base address.
- trimValue – The frequency trim value.

static inline void CCM32K_Disable32kFroIFRLoad(CCM32K_Type *base, bool disable)

Disable/Enable the function of setting 32kHz free-running oscillator trim value when IFR value gets loaded in the SOC.

Parameters

- base – CCM32K peripheral base address.
- disable – Boolean value to disable or enable IFR loading function. true — Disable IFR loading function. false — Enable IFR loading function.

static inline void CCM32K_Lock32kFroTrimWriteAccess(CCM32K_Type *base)

Lock all further write accesses to the FRO32K_TRIM register until a POR occurs.

Parameters

- base – CCM32K peripheral base address.

void CCM32K_Set32kOscConfig(CCM32K_Type *base, *ccm32k_osc_mode_t* mode, const *ccm32k_osc_config_t* *config)

Config 32k Crystal Oscillator.

Note: When the mode selected as kCCM32K_Disable32kHzCrystalOsc or kCCM32K_Bypass32kHzCrystalOsc the parameter config is useless, so it can be set as

"NULL".

Note: To enable OSC32K and select it as clock source of 32kHz please follow steps:

```
CCM32K_Set32kOscConfig(base, kCCM32K_Enable32kHzCrystalOsc, config); //Enable OSC32k and␣
↪set config.
while((CCM32K_GetStatus(base) & kCCM32K_32kOscReadyStatusFlag) == 0UL); //Check if␣
↪OSC32K is stable.
CCM32K_DisableCLKOutToPeripherals(base, mask); //Disable clock out.
CCM32K_SelectClockSource(base, kCCM32K_ClockSourceSelectOsc32k); //Select OSC32k as clock␣
↪source.
while((CCM32K_GetStatus(base) & kCCM32K_32kOscActiveStatusFlag) == 0UL); //Check if␣
↪OSC32K is used as clock source.
CCM32K_EnableCLKOutToPeripherals(base, mask); //Enable clock out.
```

Parameters

- base – CCM32K peripheral base address.
- mode – The mode of 32k crystal oscillator.
- config – The pointer to the structure ccm32k_osc_config_t.

static inline void CCM32K_Lock32kOscWriteAccess(CCM32K_Type *base)

Lock all further write accesses to the OSC32K_CTRL register until a POR occurs.

Parameters

- base – CCM32K peripheral base address.

void CCM32K_EnableClockMonitor(CCM32K_Type *base, bool enable)

Enable/disable clock monitor.

Parameters

- base – CCM32K peripheral base address.
- enable – Used to enable/disable clock monitor.
  - **turn** Enable clock monitor.
  - **false** Disable clock monitor.

static inline void CCM32K_SetClockMonitorFreqTrimValue(CCM32K_Type *base, *ccm32k_clock_monitor_freq_trim_value_t* trimValue)

Set clock monitor frequency trim value.

Parameters

- base – CCM32K peripheral base address.
- trimValue – Clock minitor frequency trim value, please refer to ccm32k_clock_monitor_freq_trim_value_t.

static inline void CCM32K_SetClockMonitorDivideTrimValue(CCM32K_Type *base, *ccm32k_clock_monitor_divide_trim_value_t* trimValue)

Set clock monitor divide trim value.

Parameters

- base – CCM32K peripheral base address.

- trimValue – Clock minitor divide trim value, please refer to ccm32k_clock_monitor_divide_trim_value_t.

void CCM32K_SetClockMonitorConfig(CCM32K_Type *base, const *ccm32k_clock_monitor_config_t* *config)

Config clock monitor one time, including frequency trim value, divide trim value.

Parameters

- base – CCM32K peripheral base address.
- config – Pointer to ccm32k_clock_monitor_config_t structure.

static inline void CCM32K_LockClockMonitorWriteAccess(CCM32K_Type *base)

Lock all further write accesses to the CLKMON_CTRL register until a POR occurs.

Parameters

- base – CCM32K peripheral base address.

static inline void CCM32K_EnableCLKOutToPeripherals(CCM32K_Type *base, uint8_t peripheralMask)

Enable 32kHz clock output to selected peripherals.

Parameters

- base – CCM32K peripheral base address.
- peripheralMask – The mask of peripherals to enable 32kHz clock output, should be the OR'ed value of ccm32k_clock_output_peripheral_t.

static inline void CCM32K_DisableCLKOutToPeripherals(CCM32K_Type *base, uint8_t peripheralMask)

Disable 32kHz clock output to selected peripherals.

Parameters

- base – CCM32K peripheral base address.
- peripheralMask – The mask of peripherals to disable 32kHz clock output, should be the OR'ed value of ccm32k_clock_output_peripheral_t.

static inline void CCM32K_SelectClockSource(CCM32K_Type *base, *ccm32k_clock_source_select_t* clockSource)

Select CCM32K module's clock source which will be provide to the device.

Parameters

- base – CCM32K peripheral base address.
- clockSource – Used to select clock source, please refer to ccm32k_clock_source_select_t for details.

static inline void CCM32K_LockClockGateWriteAccess(CCM32K_Type *base)

Lock all further write access to the CGC32K register until a POR occurs.

Parameters

- base – CCM32K peripheral base address.

static inline uint32_t CCM32K_GetStatusFlag(CCM32K_Type *base)

Get the status flag.

Parameters

- base – CCM32K peripheral base address.

Returns
The status flag of the current node. The enumerator of status flags have been provided, please see the Enumerations title for details.

ccm32k_state_t CCM32K_GetCurrentState(CCM32K_Type *base)

Get current state.

Parameters

- base – CCM32K peripheral base address.

Returns
The CCM32K's current state, please refer to ccm32k_state_t for details.

ccm32k_clock_source_t CCM32K_GetClockSource(CCM32K_Type *base)

Return current clock source.

Parameters

- base – CCM32K peripheral base address.

Return values

- kCCM32K_ClockSourceNone – The none clock source is selected.
- kCCM32K_ClockSource32kFro – 32kHz free-running oscillator is selected as clock source.
- kCCM32K_ClockSource32kOsc – 32kHz crystal oscillator is selected as clock source..

FSL_CCM32K_DRIVER_VERSION

CCM32K driver version 2.2.1.

enum _ccm32k_osc_xtal_cap

The enumerator of internal capacitance of OSC's XTAL pin.

Values:

enumerator kCCM32K_OscXtal0pFCap
The internal capacitance for XTAL pin is 0pF.

enumerator kCCM32K_OscXtal2pFCap
The internal capacitance for XTAL pin is 2pF.

enumerator kCCM32K_OscXtal4pFCap
The internal capacitance for XTAL pin is 4pF.

enumerator kCCM32K_OscXtal6pFCap
The internal capacitance for XTAL pin is 6pF.

enumerator kCCM32K_OscXtal8pFCap
The internal capacitance for XTAL pin is 8pF.

enumerator kCCM32K_OscXtal10pFCap
The internal capacitance for XTAL pin is 10pF.

enumerator kCCM32K_OscXtal12pFCap
The internal capacitance for XTAL pin is 12pF.

enumerator kCCM32K_OscXtal14pFCap
The internal capacitance for XTAL pin is 14pF.

enumerator kCCM32K_OscXtal16pFCap
The internal capacitance for XTAL pin is 16pF.

enumerator kCCM32K_OscXtal18pFCap
The internal capacitance for XTAL pin is 18pF.

enumerator kCCM32K_OscXtal20pFCap
The internal capacitance for XTAL pin is 20pF.

enumerator kCCM32K_OscXtal22pFCap
The internal capacitance for XTAL pin is 22pF.

enumerator kCCM32K_OscXtal24pFCap
The internal capacitance for XTAL pin is 24pF.

enumerator kCCM32K_OscXtal26pFCap
The internal capacitance for XTAL pin is 26pF.

enumerator kCCM32K_OscXtal28pFCap
The internal capacitance for XTAL pin is 28pF.

enumerator kCCM32K_OscXtal30pFCap
The internal capacitance for XTAL pin is 30pF.

enum _ccm32k_osc_extal_cap
The enumerator of internal capacitance of OSC's EXTAL pin.

Values:

enumerator kCCM32K_OscExtal0pFCap
The internal capacitance for EXTAL pin is 0pF.

enumerator kCCM32K_OscExtal2pFCap
The internal capacitance for EXTAL pin is 2pF.

enumerator kCCM32K_OscExtal4pFCap
The internal capacitance for EXTAL pin is 4pF.

enumerator kCCM32K_OscExtal6pFCap
The internal capacitance for EXTAL pin is 6pF.

enumerator kCCM32K_OscExtal8pFCap
The internal capacitance for EXTAL pin is 8pF.

enumerator kCCM32K_OscExtal10pFCap
The internal capacitance for EXTAL pin is 10pF.

enumerator kCCM32K_OscExtal12pFCap
The internal capacitance for EXTAL pin is 12pF.

enumerator kCCM32K_OscExtal14pFCap
The internal capacitance for EXTAL pin is 14pF.

enumerator kCCM32K_OscExtal16pFCap
The internal capacitance for EXTAL pin is 16pF.

enumerator kCCM32K_OscExtal18pFCap
The internal capacitance for EXTAL pin is 18pF.

enumerator kCCM32K_OscExtal20pFCap
The internal capacitance for EXTAL pin is 20pF.

enumerator kCCM32K_OscExtal22pFCap
The internal capacitance for EXTAL pin is 22pF.

enumerator kCCM32K_OscExtal24pFCap
The internal capacitance for EXTAL pin is 24pF.

enumerator kCCM32K_OscExtal26pFCap
The internal capacitance for EXTAL pin is 26pF.

enumerator kCCM32K_OscExtal28pFCap
The internal capacitance for EXTAL pin is 28pF.

enumerator kCCM32K_OscExtal30pFCap
The internal capacitance for EXTAL pin is 30pF.

enum _ccm32k_osc_fine_adjustment_value
The enumerator of osc amplifier gain fine adjustment. Changes the oscillator amplitude by modifying the automatic gain control (AGC).

Values:

enumerator kCCM32K_OscFineAdjustmentRange0

enum _ccm32k_osc_coarse_adjustment_value
The enumerator of osc amplifier coarse fine adjustment. Tunes the internal transconductance (gm) by increasing the current.

Values:

enumerator kCCM32K_OscCoarseAdjustmentRange0

enumerator kCCM32K_OscCoarseAdjustmentRange1

enumerator kCCM32K_OscCoarseAdjustmentRange2

enumerator kCCM32K_OscCoarseAdjustmentRange3

enum _ccm32k_osc_mode
The enumerator of 32kHz oscillator.

Values:

enumerator kCCM32K_Disable32kHzCrystalOsc
Disable 32kHz Crystal Oscillator.

enumerator kCCM32K_Enable32kHzCrystalOsc
Enable 32kHz Crystal Oscillator.

enumerator kCCM32K_Bypass32kHzCrystalOsc
Bypass 32kHz Crystal Oscillator, use the 32kHz Oscillator or external 32kHz clock.

The enumerator of CCM32K status flag.

Values:

enumerator kCCM32K_32kOscReadyStatusFlag
Indicates the 32kHz crystal oscillator is stable.

enumerator kCCM32K_32kOscActiveStatusFlag
Indicates the 32kHz crystal oscillator is active and in use.

enumerator kCCM32K_32kFroActiveStatusFlag
Indicates the 32kHz free running oscillator is active and in use.

enumerator kCCM32K_ClockDetectStatusFlag
Indicates the clock monitor has detected an error.

enum _ccm32k_state

The enumerator of module state.

Values:

enumerator kCCM32K_Both32kFro32kOscDisabled

Indicates both 32kHz free running oscillator and 32kHz crystal oscillator are disabled.

enumerator kCCM32K_Only32kFroEnabled

Indicates only 32kHz free running oscillator is enabled.

enumerator kCCM32K_Only32kOscEnabled

Indicates only 32kHz crystal oscillator is enabled.

enumerator kCCM32K_Both32kFro32kOscEnabled

Indicates both 32kHz free running oscillator and 32kHz crystal oscillator are enabled.

enum _ccm32k_clock_source

The enumerator of clock source.

Values:

enumerator kCCM32K_ClockSourceNone

None clock source.

enumerator kCCM32K_ClockSource32kFro

32kHz free running oscillator is the clock source.

enumerator kCCM32K_ClockSource32kOsc

32kHz crystal oscillator is the clock source.

enum _ccm32k_clock_monitor_freq_trim_value

Clock monitor frequency trim values.

Values:

enumerator kCCM32K_ClockMonitor2CycleAssert

Clock monitor asserts 2 cycle after expected edge (assert after 10 cycles with no edge).

enumerator kCCM32K_ClockMonitor4CycleAssert

Clock monitor asserts 4 cycle after expected edge (assert after 12 cycles with no edge).

enumerator kCCM32K_ClockMonitor6CycleAssert

Clock monitor asserts 6 cycle after expected edge (assert after 14 cycles with no edge).

enumerator kCCM32K_ClockMonitor8CycleAssert

Clock monitor asserts 8 cycle after expected edge (assert after 16 cycles with no edge).

enum _ccm32k_clock_monitor_divide_trim_value

Clock monitor divide trim values.

Values:

enumerator kCCM32K_ClockMonitor_1kHzFro32k_1kHzOsc32k

Clock monitor operates at 1 kHz for both FRO32K and OSC32K.

enumerator kCCM32K_ClockMonitor_64HzFro32k_1kHzOsc32k

Clock monitor operates at 64 Hz for FRO32K and clock monitor operates at 1 kHz for OSC32K.

enumerator kCCM32K_ClockMonitor_1KHzFro32k_64HzOsc32k

Clock monitor operates at 1K Hz for FRO32K and clock monitor operates at 64 Hz for OSC32K.

enumerator kCCM32K_ClockMonitor_64HzFro32k_64HzOsc32k
Clock monitor operates at 64 Hz for FRO32K and clock monitor operates at 64 Hz for OSC32K.

enum _ccm32k_clock_source_select
CCM32K clock source enumeration.

Values:

enumerator kCCM32K_ClockSourceSelectFro32k
FRO32K clock output is selected as clock source.

enumerator kCCM32K_ClockSourceSelectOsc32k
OSC32K clock output is selected as clock source.

enum _ccm32k_clock_output_peripheral
32kHz clock output peripheral bit map.

Values:

enumerator kCCM32K_ClockOutToRtc
32kHz clock output to RTC.

enumerator kCCM32K_ClockOutToRfmc
32kHz clock output to Rfmc.

enumerator kCCM32K_ClockOutToNbu
32kHz clock output to NBU.

enumerator kCCM32K_ClockOutToWuuRmcPortD
32kHz clock output to WUU/RMC/PORTD.

enumerator kCCM32K_ClockOutToOtherModules
32kHz clock output to Other modules.

typedef enum *_ccm32k_osc_xtal_cap* ccm32k_osc_xtal_cap_t
The enumerator of internal capacitance of OSC's XTAL pin.

typedef enum *_ccm32k_osc_extal_cap* ccm32k_osc_extal_cap_t
The enumerator of internal capacitance of OSC's EXTAL pin.

typedef enum *_ccm32k_osc_fine_adjustment_value* ccm32k_osc_fine_adjustment_value_t
The enumerator of osc amplifier gain fine adjustment. Changes the oscillator amplitude by modifying the automatic gain control (AGC).

typedef enum *_ccm32k_osc_coarse_adjustment_value* ccm32k_osc_coarse_adjustment_value_t
The enumerator of osc amplifier coarse fine adjustment. Tunes the internal transconductance (gm) by increasing the current.

typedef enum *_ccm32k_osc_mode* ccm32k_osc_mode_t
The enumerator of 32kHz oscillator.

typedef enum *_ccm32k_state* ccm32k_state_t
The enumerator of module state.

typedef enum *_ccm32k_clock_source* ccm32k_clock_source_t
The enumerator of clock source.

typedef enum *_ccm32k_clock_monitor_freq_trim_value* ccm32k_clock_monitor_freq_trim_value_t
Clock monitor frequency trim values.

typedef enum *_ccm32k_clock_monitor_divide_trim_value* ccm32k_clock_monitor_divide_trim_value_t

Clock monitor divide trim values.

typedef struct *_ccm32k_clock_monitor_config* ccm32k_clock_monitor_config_t

Clock monitor configuration structure.

typedef enum *_ccm32k_clock_source_select* ccm32k_clock_source_select_t

CCM32K clock source enumeration.

typedef enum *_ccm32k_clock_output_peripheral* ccm32k_clock_output_peripheral_t

32kHz clock output peripheral bit map.

typedef struct *_ccm32k_osc_config* ccm32k_osc_config_t

The structure of oscillator configuration.

CCM32K_OSC32K_CTRL_OSC_MODE_MASK

CCM32K_OSC32K_CTRL_OSC_MODE_SHIFT

CCM32K_OSC32K_CTRL_OSC_MODE(**x**)

struct _ccm32k_clock_monitor_config

#include <fsl_ccm32k.h> Clock monitor configuration structure.

Public Members

bool enableClockMonitor

Used to enable/disable clock monitor.

ccm32k_clock_monitor_freq_trim_value_t freqTrimValue

Clock minitor frequency trim value.

ccm32k_clock_monitor_divide_trim_value_t divideTrimValue

Clock minitor divide trim value.

struct _ccm32k_osc_config

#include <fsl_ccm32k.h> The structure of oscillator configuration.

Public Members

bool enableInternalCapBank

enable/disable the internal capacitance bank.

ccm32k_osc_xtal_cap_t xtalCap

The internal capacitance for the OSC XTAL pin from the capacitor bank, only useful when the internal capacitance bank is enabled.

ccm32k_osc_extal_cap_t extalCap

The internal capacitance for the OSC EXTAL pin from the capacitor bank, only useful when the internal capacitance bank is enabled.

ccm32k_osc_fine_adjustment_value_t fineAdjustment

32kHz crystal oscillator amplifier fine adjustment value.

ccm32k_osc_coarse_adjustment_value_t coarseAdjustment

32kHz crystal oscillator amplifier coarse adjustment value.

2.3 Clock Driver

uint32_t CLOCK_GetSysClkFreq(*scg_sys_clk_t* type)

Gets the SCG system clock frequency.

This function gets the SCG system clock frequency. These clocks are used for core, platform, external, and bus clock domains.

Parameters

- type – Which type of clock to get, core clock or slow clock.

Returns

Clock frequency.

static inline void CLOCK_SetRunModeSysClkConfig(const *scg_sys_clk_config_t* *config)

Sets the system clock configuration for RUN mode.

This function sets the system clock configuration for RUN mode.

Parameters

- config – Pointer to the configuration.

static inline void CLOCK_GetCurSysClkConfig(*scg_sys_clk_config_t* *config)

Gets the system clock configuration in the current power mode.

This function gets the system configuration in the current power mode.

Parameters

- config – Pointer to the configuration.

static inline void CLOCK_SetClkOutSel(*clock_clkout_src_t* setting)

Sets the clock out selection.

This function sets the clock out selection (CLKOUTSEL).

Parameters

- setting – The selection to set.

status_t CLOCK_InitSysOsc(const *scg_sosc_config_t* *config)

Initializes the SCG system OSC.

This function enables the SCG system OSC clock according to the configuration.

Note: This function can't detect whether the system OSC has been enabled and used by an IP.

Parameters

- config – Pointer to the configuration structure.

Return values

- kStatus_Success – System OSC is initialized.
- kStatus_SCG_Busy – System OSC has been enabled and is used by the system clock.
- kStatus_ReadOnly – System OSC control register is locked.

status_t CLOCK_DeinitSysOsc(void)

De-initializes the SCG system OSC.

This function disables the SCG system OSC clock.

Note: This function can't detect whether the system OSC is used by an IP.

Return values

- kStatus_Success – System OSC is deinitialized.
- kStatus_SCG_Busy – System OSC is used by the system clock.
- kStatus_ReadOnly – System OSC control register is locked.

uint32_t CLOCK_GetSysOscFreq(void)

Gets the SCG system OSC clock frequency (SYSOSC).

Returns

Clock frequency; If the clock is invalid, returns 0.

static inline bool CLOCK_IsSysOscErr(void)

Checks whether the system OSC clock error occurs.

Returns

True if the error occurs, false if not.

static inline void CLOCK_ClearSysOscErr(void)

Clears the system OSC clock error.

static inline void CLOCK_SetSysOscMonitorMode(*scg_sosc_monitor_mode_t* mode)

Sets the system OSC monitor mode.

This function sets the system OSC monitor mode. The mode can be disabled, it can generate an interrupt when the error is disabled, or reset when the error is detected.

Parameters

- mode – Monitor mode to set.

static inline bool CLOCK_IsSysOscValid(void)

Checks whether the system OSC clock is valid.

Returns

True if clock is valid, false if not.

static inline void CLOCK_UnlockSysOscControlStatusReg(void)

Unlock the SOSCCSR control status register.

static inline void CLOCK_LockSysOscControlStatusReg(void)

Lock the SOSCCSR control status register.

status_t CLOCK_InitSirc(const *scg_sirc_config_t* *config)

Initializes the SCG slow IRC clock.

This function enables the SCG slow IRC clock according to the configuration.

Note: This function can't detect whether the system OSC has been enabled and used by an IP.

Parameters

- config – Pointer to the configuration structure.

Return values

- kStatus_Success – SIRC is initialized.
- kStatus_SCG_Busy – SIRC has been enabled and is used by system clock.
- kStatus_ReadOnly – SIRC control register is locked.

status_t CLOCK_DeinitSirc(void)

De-initializes the SCG slow IRC.

This function disables the SCG slow IRC.

Note: This function can't detect whether the SIRC is used by an IP.

Return values

- kStatus_Success – SIRC is deinitialized.
- kStatus_SCG_Busy – SIRC is used by system clock.
- kStatus_ReadOnly – SIRC control register is locked.

uint32_t CLOCK_GetSircFreq(void)

Gets the SCG SIRC clock frequency.

Returns
Clock frequency; If the clock is invalid, returns 0.

static inline bool CLOCK_IsSircValid(void)

Checks whether the SIRC clock is valid.

Returns
True if clock is valid, false if not.

static inline void CLOCK_UnlockSircControlStatusReg(void)

Unlock the SIRCCSR control status register.

static inline void CLOCK_LockSircControlStatusReg(void)

Lock the SIRCCSR control status register.

status_t CLOCK_InitFirc(const *scg_firc_config_t* *config)

Initializes the SCG fast IRC clock.

This function enables the SCG fast IRC clock according to the configuration.

Note: This function can't detect whether the FIRC has been enabled and used by an IP.

Parameters

- config – Pointer to the configuration structure.

Return values

- kStatus_Success – FIRC is initialized.
- kStatus_SCG_Busy – FIRC has been enabled and is used by the system clock.
- kStatus_ReadOnly – FIRC control register is locked.

status_t CLOCK_DeinitFirc(void)

De-initializes the SCG fast IRC.

This function disables the SCG fast IRC.

Note: This function can't detect whether the FIRC is used by an IP.

Return values

- kStatus_Success – FIRC is deinitialized.
- kStatus_SCG_Busy – FIRC is used by the system clock.
- kStatus_ReadOnly – FIRC control register is locked.

uint32_t CLOCK_GetFircFreq(void)

Gets the SCG FIRC clock frequency.

Returns
Clock frequency; If the clock is invalid, returns 0.

static inline bool CLOCK_IsFircErr(void)

Checks whether the FIRC clock error occurs.

Returns
True if the error occurs, false if not.

static inline void CLOCK_ClearFircErr(void)

Clears the FIRC clock error.

static inline bool CLOCK_IsFircValid(void)

Checks whether the FIRC clock is valid.

Returns
True if clock is valid, false if not.

static inline void CLOCK_UnlockFircControlStatusReg(void)

Unlock the FIRCCSR control status register.

static inline void CLOCK_LockFircControlStatusReg(void)

Lock the FIRCCSR control status register.

static inline bool CLOCK_IsFIRCAutoTrimLocked(void)

Check whether FIRC auto trim locked to target frequency range.

When FIRCTREN and FIRCTRUP are enabled, TRIM_LOCK will indicate when auto trimming is complete and output FIRC frequency has locked to target FIRC range. TRIM_LOCK will automatically get cleared if FIRCTREN and FIRCTRUP are not set.

Returns
True if FIRC trim locked to target frequency range, false if not.

status_t CLOCK_InitRosc(const *scg_rosc_config_t* *config)

brief Initializes the SCG ROSC.

This function enables the SCG ROSC clock according to the configuration.

param config Pointer to the configuration structure. retval kStatus_Success ROSC is initialized. retval kStatus_SCG_Busy ROSC has been enabled and is used by the system clock. retval kStatus_ReadOnly ROSC control register is locked.

note This function can't detect whether the system OSC has been enabled and used by an IP.

status_t CLOCK_DeinitRosc(void)

brief De-initializes the SCG ROSC.

This function disables the SCG ROSC clock.

retval kStatus_Success System OSC is deinitialized. retval kStatus_SCG_Busy System OSC is used by the system clock. retval kStatus_ReadOnly System OSC control register is locked.

note This function can't detect whether the ROSC is used by an IP.

uint32_t CLOCK_GetRtcOscFreq(void)

Gets the SCG RTC OSC clock frequency.

Returns
Clock frequency; If the clock is invalid, returns 0.

status_t CLOCK_InitRfFro192M(const *fro192m_rf_clk_config_t* *config)

Initializes the FRO192M clock for the Radio Mode Controller.

This function configure the RF FRO192M clock according to the configuration.

Parameters

- config – Pointer to the configuration structure.

Return values
kStatus_Success – RF FRO192M is configured.

uint32_t CLOCK_GetRfFro192MFreq(void)

Gets the FRO192M clock frequency.

Returns
Clock frequency; If the clock is invalid, returns 0.

static inline bool CLOCK_IsRoscErr(void)

Checks whether the ROSC clock error occurs.

Returns
True if the error occurs, false if not.

static inline void CLOCK_ClearRoscErr(void)

Clears the ROSC clock error.

static inline void CLOCK_SetRoscMonitorMode(*scg_rosc_monitor_mode_t* mode)

Sets the ROSC monitor mode.

This function sets the ROSC monitor mode. The mode can be disabled, it can generate an interrupt when the error is disabled, or reset when the error is detected.

Parameters

- mode – Monitor mode to set.

static inline bool CLOCK_IsRoscValid(void)

Checks whether the ROSC clock is valid.

Returns
True if clock is valid, false if not.

static inline void CLOCK_UnlockRoscControlStatusReg(void)

Unlock the ROSCCSR control status register.

static inline void CLOCK_LockRoscControlStatusReg(void)

Lock the ROSCCSR control status register.

static inline void CLOCK_SetXtal0Freq(uint32_t freq)

Sets the XTAL0 frequency based on board settings.

Parameters

- freq – The XTAL0/EXTAL0 input clock frequency in Hz.

static inline void CLOCK_SetXtal32Freq(uint32_t freq)

Sets the XTAL32 frequency based on board settings.

Parameters

- freq – The XTAL32/EXTAL32 input clock frequency in Hz.

FSL_CLOCK_DRIVER_VERSION

CLOCK driver version 2.2.5.

enum _clock_name

Clock name used to get clock frequency.

These clocks source would be generated from SCG module.

Values:

enumerator kCLOCK_CoreSysClk

Cortex M33 clock.

enumerator kCLOCK_SlowClk

SLOW_CLK with DIVSLOW.

enumerator kCLOCK_PlatClk

PLAT_CLK.

enumerator kCLOCK_SysClk

SYS_CLK.

enumerator kCLOCK_BusClk

BUS_CLK with DIVBUS.

enumerator kCLOCK_ScgSysOscClk

SCG system OSC clock.

enumerator kCLOCK_ScgSircClk

SCG SIRC clock.

enumerator kCLOCK_ScgFircClk

SCG FIRC clock.

enumerator kCLOCK_RtcOscClk

RTC OSC clock.

enum _clock_ip_control

Clock source for peripherals that support various clock selections.

These options are for MRCC->XX[CC]

Values:

enumerator kCLOCK_IpClkControl_fun0

Peripheral clocks are disabled, module does not stall low power mode entry.

enumerator kCLOCK_IpClkControl_fun1

Peripheral clocks are enabled, module does not stall low power mode entry.

enumerator kCLOCK_IpClkControl_fun2
Peripherals clocks are enabled unless peripheral is idle, low power mode entry will stall until peripheral is idle.

enumerator kCLOCK_IpClkControl_fun3
Peripheral clocks are enabled unless in SLEEP mode (or lower), low power mode entry will stall until peripheral is idle Peripheral functional clocks that remain enabled in SLEEP mode are enabled and do not stall low power mode entry unless entering DEEPSLEEP mode (or lower)

enum _clock_ip_src
Clock source for peripherals that support various clock selections.

These options are for MRCC->XX[MUX].

Values:

enumerator kCLOCK_IpSrcFro6M
FRO 6M clock.

enumerator kCLOCK_IpSrcFro192M
FRO 192M clock.

enumerator kCLOCK_IpSrcSoscClk
OSC RF clock.

enumerator kCLOCK_IpSrc32kClk
32k Clk clock.

enum _tpm2_ip_src
Clock source for TPM2.

These options are for RF_CMC1->TPM2_CFG[CLK_MUX_SEL].

Values:

enumerator kCLOCK_Tpm2SrcCoreClk
Core Clock.

enumerator kCLOCK_Tpm2SrcSoscClk
Radio Oscillator.

enum _clock_ip_name
Clock IP name.

Values:

enumerator kCLOCK_NOGATE
No clock gate for the IP in MRCC

enumerator kCLOCK_Ewm0
Clock ewm0

enumerator kCLOCK_Syspm0
Clock syspm0

enumerator kCLOCK_Wdog0
Clock wdog0

enumerator kCLOCK_Wdog1
Clock wdog1

enumerator kCLOCK_Sfa0
Clock sfa0

enumerator kCLOCK_Crc0
Clock crc0

enumerator kCLOCK_Secsubsys
Clock secsubsys

enumerator kCLOCK_Lpit0
Clock lpit0

enumerator kCLOCK_Tstmr0
Clock tstmr0

enumerator kCLOCK_Tpm0
Clock tpm0

enumerator kCLOCK_Tpm1
Clock tpm1

enumerator kCLOCK_Lpi2c0
Clock lpi2c0

enumerator kCLOCK_Lpi2c1
Clock lpi2c1

enumerator kCLOCK_I3c0
Clock i3c

enumerator kCLOCK_Lpspi0
Clock lpspi0

enumerator kCLOCK_Lpspi1
Clock lpspi1

enumerator kCLOCK_Lpuart0
Clock lpuart0

enumerator kCLOCK_Lpuart1
Clock lpuart1

enumerator kCLOCK_Flexio0
Clock Flexio0

enumerator kCLOCK_Can0
Clock Can0

enumerator kCLOCK_Sema0
Clock Sema0

enumerator kCLOCK_Data_stream_2p4
Clock data_stream_2p4

enumerator kCLOCK_PortA
Clock portA

enumerator kCLOCK_PortB
Clock portB

enumerator kCLOCK_PortC
Clock portC

enumerator kCLOCK_Lpadc0
Clock lpadc0

enumerator kCLOCK_Lpcmp0
Clock lpcmp0

enumerator kCLOCK_Lpcmp1
Clock lpcmp1

enumerator kCLOCK_Vref0
Clock verf0

enumerator kCLOCK_Mtr_master
Clock mtr_master

enumerator kCLOCK_GpioA
Clock gpioA

enumerator kCLOCK_GpioB
Clock gpioB

enumerator kCLOCK_GpioC
Clock gpioC

enumerator kCLOCK_Dma0
Clock dma0

enumerator kCLOCK_Pflexnvm
Clock pflexnvm

enumerator kCLOCK_Sram0
Clock Sram0

enumerator kCLOCK_Sram1
Clock Sram1

enumerator kCLOCK_Sram2
Clock Sram2

enumerator kCLOCK_Sram3
Clock Sram3

enumerator kCLOCK_Rf_2p4ghz_bist
Clock rf_2p4ghz_bist

enum _scg_status
SCG status return codes.

Values:

enumerator kStatus_SCG_Busy
Clock is busy.

enumerator kStatus_SCG_InvalidSrc
Invalid source.

enum _scg_sys_clk
SCG system clock type.

Values:

enumerator kSCG_SysClkSlow
System slow clock.

enumerator kSCG_SysClkBus
Bus clock.

enumerator kSCG_SysClkPlatform
Platform clock.

enumerator kSCG_SysClkCore
Core clock.

enum _scg_sys_clk_src
SCG system clock source.

ERR052742: FRO6M clock(kSCG_SysClkSrcSirc) is not stable. The FRO6M clock is not stable on some parts. FRO6M outputs lower frequency signal instead of 6MHz when device is reset or wakes up from low power. It can impact peripherals using it as a clock source. Please use clock source other than the FRO6M. For example, use FRO192M instead of FRO6M as clock source for peripherals.

Values:

enumerator kSCG_SysClkSrcSysOsc
System OSC.

enumerator kSCG_SysClkSrcSirc
Slow IRC.

enumerator kSCG_SysClkSrcFirc
Fast IRC.

enumerator kSCG_SysClkSrcRosc
RTC OSC.

enum _scg_sys_clk_div
SCG system clock divider value.

Values:

enumerator kSCG_SysClkDivBy1
Divided by 1.

enumerator kSCG_SysClkDivBy2
Divided by 2.

enumerator kSCG_SysClkDivBy3
Divided by 3.

enumerator kSCG_SysClkDivBy4
Divided by 4.

enumerator kSCG_SysClkDivBy5
Divided by 5.

enumerator kSCG_SysClkDivBy6
Divided by 6.

enumerator kSCG_SysClkDivBy7
Divided by 7.

enumerator kSCG_SysClkDivBy8
Divided by 8.

enumerator kSCG_SysClkDivBy9
Divided by 9.

enumerator kSCG_SysClkDivBy10
Divided by 10.

enumerator kSCG_SysClkDivBy11
Divided by 11.

enumerator kSCG_SysClkDivBy12
Divided by 12.

enumerator kSCG_SysClkDivBy13
Divided by 13.

enumerator kSCG_SysClkDivBy14
Divided by 14.

enumerator kSCG_SysClkDivBy15
Divided by 15.

enumerator kSCG_SysClkDivBy16
Divided by 16.

enum _clock_clkout_src
SCG clock out configuration (CLKOUTSEL).

Values:

enumerator kClockClkoutSelScgSlow
SCG Slow clock.

enumerator kClockClkoutSelSosc
System OSC.

enumerator kClockClkoutSelSirc
Slow IRC.

enumerator kClockClkoutSelFirc
Fast IRC.

enumerator kClockClkoutSelScgRtcOsc
SCG RTC OSC clock.

enum _scg_sosc_monitor_mode
SCG system OSC monitor mode.

Values:

enumerator kSCG_SysOscMonitorDisable
Monitor disabled.

enumerator kSCG_SysOscMonitorInt
Interrupt when the SOSC error is detected.

enumerator kSCG_SysOscMonitorReset
Reset when the SOSC error is detected.

SOSC enable mode.

Values:

enumerator kSCG_SoscDisable
Disable SOSC clock.

enumerator kSCG_SoscEnable
Enable SOSC clock.

enumerator kSCG_SoscEnableInSleep
Enable SOSC in sleep mode.

enum _scg_rosc_monitor_mode
SCG ROSC monitor mode.

Values:

enumerator kSCG_RoscMonitorDisable
Monitor disabled.

enumerator kSCG_RoscMonitorInt
Interrupt when the RTC OSC error is detected.

enumerator kSCG_RoscMonitorReset
Reset when the RTC OSC error is detected.

enum _scg_sirc_enable_mode
SIRC enable mode.

Values:

enumerator kSCG_SircDisableInSleep
Disable SIRC clock.

enumerator kSCG_SircEnableInSleep
Enable SIRC in sleep mode.

enum _scg_firc_trim_mode
SCG fast IRC trim mode.

Values:

enumerator kSCG_FircTrimNonUpdate
FIRC trim enable but not enable trim value update. In this mode, the trim value is fixed to the initialized value which is defined by trimCoar and trimFine in configure structure scg_firc_trim_config_t.

enumerator kSCG_FircTrimUpdate
FIRC trim enable and trim value update enable. In this mode, the trim value is auto update.

enum _scg_firc_trim_src
SCG fast IRC trim source.

Values:

enumerator kSCG_FircTrimSrcSysOsc
System OSC.

enumerator kSCG_FircTrimSrcRtcOsc
RTC OSC (32.768 kHz).

FIRC enable mode.

Values:

enumerator kSCG_FircDisable
Disable FIRC clock.

enumerator kSCG_FircEnable
Enable FIRC clock.

enumerator kSCG_FircEnableInSleep
Enable FIRC in sleep mode.

enum _scg_firc_range
SCG fast IRC clock frequency range.

Values:

enumerator kSCG_FircRange48M
Fast IRC is trimmed to 48 MHz.

enumerator kSCG_FircRange64M
Fast IRC is trimmed to 64 MHz.

enumerator kSCG_FircRange96M
Fast IRC is trimmed to 96 MHz.

enumerator kSCG_FircRange192M
Fast IRC is trimmed to 192 MHz.

enum _fro192m_rf_range
FRO192M RF clock frequency range.

Values:

enumerator kFro192M_Range16M
FRO192M output frequenc 16 MHz.

enumerator kFro192M_Range24M
FRO192M output frequenc 24 MHz.

enumerator kFro192M_Range32M
FRO192M output frequenc 32 MHz.

enumerator kFro192M_Range48M
FRO192M output frequenc 48 MHz.

enumerator kFro192M_Range64M
FRO192M output frequenc 64 MHz.

enum _fro192m_rf_clk_div
RF Flash APB and RF_CMC clock divide.

Values:

enumerator kFro192M_ClkDivBy1
Divided by 1.

enumerator kFro192M_ClkDivBy2
Divided by 2.

enumerator kFro192M_ClkDivBy4
Divided by 4.

enumerator kFro192M_ClkDivBy8
Divided by 8.

typedef enum *_clock_name* clock_name_t
Clock name used to get clock frequency.

These clocks source would be generated from SCG module.

typedef enum *_clock_ip_control* clock_ip_control_t

Clock source for peripherals that support various clock selections.

These options are for MRCC->XX[CC]

typedef enum *_clock_ip_src* clock_ip_src_t

Clock source for peripherals that support various clock selections.

These options are for MRCC->XX[MUX].

typedef enum *_tpm2_ip_src* tpm2_src_t

Clock source for TPM2.

These options are for RF_CMC1->TPM2_CFG[CLK_MUX_SEL].

typedef enum *_clock_ip_name* clock_ip_name_t

Clock IP name.

typedef enum *_scg_sys_clk* scg_sys_clk_t

SCG system clock type.

typedef enum *_scg_sys_clk_src* scg_sys_clk_src_t

SCG system clock source.

ERR052742: FRO6M clock(kSCG_SysClkSrcSirc) is not stable. The FRO6M clock is not stable on some parts. FRO6M outputs lower frequency signal instead of 6MHz when device is reset or wakes up from low power. It can impact peripherals using it as a clock source. Please use clock source other than the FRO6M. For example, use FRO192M instead of FRO6M as clock source for peripherals.

typedef enum *_scg_sys_clk_div* scg_sys_clk_div_t

SCG system clock divider value.

typedef struct *_scg_sys_clk_config* scg_sys_clk_config_t

SCG system clock configuration.

typedef enum *_clock_clkout_src* clock_clkout_src_t

SCG clock out configuration (CLKOUTSEL).

typedef enum *_scg_sosc_monitor_mode* scg_sosc_monitor_mode_t

SCG system OSC monitor mode.

typedef struct *_scg_sosc_config* scg_sosc_config_t

SCG system OSC configuration.

typedef enum *_scg_rosc_monitor_mode* scg_rosc_monitor_mode_t

SCG ROSC monitor mode.

typedef struct *_scg_rosc_config* scg_rosc_config_t

SCG ROSC configuration.

typedef enum *_scg_sirc_enable_mode* scg_sirc_enable_mode_t

SIRC enable mode.

typedef struct *_scg_sirc_config* scg_sirc_config_t

SCG slow IRC clock configuration.

typedef enum *_scg_firc_trim_mode* scg_firc_trim_mode_t

SCG fast IRC trim mode.

typedef enum *_scg_firc_trim_src* scg_firc_trim_src_t

SCG fast IRC trim source.

typedef struct *_scg_firc_trim_config* scg_firc_trim_config_t

SCG fast IRC clock trim configuration.

typedef enum *_scg_firc_range* scg_firc_range_t

SCG fast IRC clock frequency range.

typedef struct *_scg_firc_config_t* scg_firc_config_t

SCG fast IRC clock configuration.

typedef enum *_fro192m_rf_range* fro192m_rf_range_t

FRO192M RF clock frequency range.

typedef enum *_fro192m_rf_clk_div* fro192m_rf_clk_div_t

RF Flash APB and RF_CMC clock divide.

typedef struct *_fro192m_rf_clk_config* fro192m_rf_clk_config_t

FRO192M RF clock configuration.

volatile uint32_t g_xtal0Freq

External XTAL0 (OSC0/SYSOSC) clock frequency.

The XTAL0/EXTAL0 (OSC0/SYSOSC) clock frequency in Hz. When the clock is set up, use the function CLOCK_SetXtal0Freq to set the value in the clock driver. For example, if XTAL0 is 8 MHz:

```
CLOCK_InitSysOsc(...);
CLOCK_SetXtal0Freq(80000000);
```

This is important for the multicore platforms where only one core needs to set up the OSC0/SYSOSC using CLOCK_InitSysOsc. All other cores need to call the CLOCK_SetXtal0Freq to get a valid clock frequency.

volatile uint32_t g_xtal32Freq

External XTAL32/EXTAL32 clock frequency.

The XTAL32/EXTAL32 clock frequency in Hz. When the clock is set up, use the function CLOCK_SetXtal32Freq to set the value in the clock driver.

This is important for the multicore platforms where only one core needs to set up the clock. All other cores need to call the CLOCK_SetXtal32Freq to get a valid clock frequency.

static inline void CLOCK_EnableClock(*clock_ip_name_t* name)

Enable the clock for specific IP.

Parameters

- name – Which clock to enable, see clock_ip_name_t.

static inline void CLOCK_EnableTPM2(void)

Enable the TPM2 clock.

static inline void CLOCK_EnableClockLPMode(*clock_ip_name_t* name, *clock_ip_control_t* control)

Enable the clock for specific IP in low power mode.

Parameters

- name – Which clock to enable, see clock_ip_name_t.
- control – Clock Config, see clock_ip_control_t.

static inline void CLOCK_DisableClock(*clock_ip_name_t* name)

Disable the clock for specific IP.

Parameters

- name – Which clock to disable, see clock_ip_name_t.

static inline void CLOCK_DisableTPM2(void)

Disable the TPM2 clock.

static inline void CLOCK_SetIpSrc(*clock_ip_name_t* name, *clock_ip_src_t* src)

Set the clock source for specific IP module.

Set the clock source for specific IP, not all modules need to set the clock source, should only use this function for the modules need source setting. ERR052742: FRO6M clock is not stable. The FRO6M clock is not stable on some parts. FRO6M outputs lower frequency signal instead of 6MHz when device is reset or wakes up from low power. It can impact peripherals using it as a clock source. Please use clock source other than the FRO6M. For example, use FRO192M instead of FRO6M as clock source for peripherals.

Parameters

- name – Which peripheral to check, see clock_ip_name_t.
- src – Clock source to set.

static inline void CLOCK_SetTpm2Src(*tpm2_src_t* src)

Set the clock source for TPM2.

Parameters

- src – Clock source to set.

static inline void CLOCK_SetIpSrcDiv(*clock_ip_name_t* name, uint8_t divValue)

Set the clock source and divider for specific IP module.

Set the clock source and divider for specific IP, not all modules need to set the clock source and divider, should only use this function for the modules need source and divider setting.

Divider output clock = Divider input clock / (divValue+1)]).

Parameters

- name – Which peripheral to check, see clock_ip_name_t.
- divValue – The divider value.

uint32_t CLOCK_GetFreq(*clock_name_t* clockName)

Gets the clock frequency for a specific clock name.

This function checks the current clock configurations and then calculates the clock frequency for a specific clock name defined in clock_name_t.

Parameters

- clockName – Clock names defined in clock_name_t

Returns

Clock frequency value in hertz

uint32_t CLOCK_GetCoreSysClkFreq(void)

Get the core clock or system clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetPlatClkFreq(void)

Get the platform clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetBusClkFreq(void)

Get the bus clock frequency.

Returns
Clock frequency in Hz.

uint32_t CLOCK_GetFlashClkFreq(void)

Get the flash clock frequency.

Returns
Clock frequency in Hz.

uint32_t CLOCK_GetIpFreq(*clock_ip_name_t* name)

Gets the functional clock frequency for a specific IP module.

This function gets the IP module's functional clock frequency based on MRCC registers. It is only used for the IP modules which could select clock source by MRCC[PCS].

Parameters

- name – Which peripheral to get, see clock_ip_name_t.

Returns
Clock frequency value in Hz

FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL

Configure whether driver controls clock.

When set to 0, peripheral drivers will enable clock in initialize function and disable clock in de-initialize function. When set to 1, peripheral driver will not control the clock, application could control the clock out of the driver.

Note: All drivers share this feature switcher. If it is set to 1, application should handle clock enable and disable for all drivers.

SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY

EDMA_CLOCKS

Clock ip name array for EDMA.

SYSPM_CLOCKS

Clock ip name array for SYSPM.

SFA_CLOCKS

Clock ip name array for SFA.

CRC_CLOCKS

Clock ip name array for CRC.

TPM_CLOCKS

Clock ip name array for TPM.

LPI2C_CLOCKS

Clock ip name array for LPI2C.

I3C_CLOCKS

Clock ip name array for I3C.

LPSPI_CLOCKS

Clock ip name array for LPSPI.

LPUART_CLOCKS

Clock ip name array for LPUART.

PORT_CLOCKS
Clock ip name array for PORT.

LPADC_CLOCKS
Clock ip name array for LPADC.

LPCMP_CLOCKS
Clock ip name array for LPCMP.

VREF_CLOCKS
Clock ip name array for VREF.

GPIO_CLOCKS
Clock ip name array for GPIO.

LPIT_CLOCKS
Clock ip name array for LPIT.

RF_CLOCKS
Clock ip name array for RF.

WDOG_CLOCKS
Clock ip name array for WDOG.

FLEXCAN_CLOCKS
Clock ip name array for FLEXCAN.

FLEXIO_CLOCKS
Clock ip name array for FLEXIO.

TSTMR_CLOCKS
Clock ip name array for TSTMR.

EWM_CLOCKS
Clock ip name array for EWM.

SEMA42_CLOCKS
Clock ip name array for SEMA42.

MAKE_MRCC_REGADDR(base, offset)
"IP Connector name difinition used for clock gate, clock source and clock divider setting. It is defined as the corresponding register address.

CLOCK_REG(name)

struct _scg_sys_clk_config
#include <fsl_clock.h> SCG system clock configuration.

Public Members

uint32_t divSlow
Slow clock divider, see scg_sys_clk_div_t.

uint32_t divBus
Bus clock divider, see scg_sys_clk_div_t.

uint32_t ___pad0___
Reserved.

uint32_t divCore
Core clock divider, see scg_sys_clk_div_t.

uint32_t ___pad1___
Reserved.

uint32_t src
System clock source, see scg_sys_clk_src_t.

uint32_t ___pad2___
reserved.

struct __scg__sosc__config
#include <fsl_clock.h> SCG system OSC configuration.

Public Members

uint32_t freq
System OSC frequency.

uint32_t enableMode
Enable mode, OR'ed value of _scg_sosc_enable_mode.

scg_sosc_monitor_mode_t monitorMode
Clock monitor mode selected.

struct __scg__rosc__config
#include <fsl_clock.h> SCG ROSC configuration.

Public Members

scg_rosc_monitor_mode_t monitorMode
Clock monitor mode selected.

struct __scg__sirc__config
#include <fsl_clock.h> SCG slow IRC clock configuration.

Public Members

scg_sirc_enable_mode_t enableMode
Enable mode, OR'ed value of _scg_sirc_enable_mode.

struct __scg__firc__trim__config
#include <fsl_clock.h> SCG fast IRC clock trim configuration.

Public Members

scg_firc_trim_mode_t trimMode
FIRC trim mode.

scg_firc_trim_src_t trimSrc
Trim source.

uint16_t trimDiv
Divider of SOSC for FIRC.

uint8_t trimCoar
Trim coarse value; Irrelevant if trimMode is kSCG_FircTrimUpdate.

uint8_t trimFine
Trim fine value; Irrelevant if trimMode is kSCG_FircTrimUpdate.

struct __scg__firc__config__t
#include <fsl_clock.h> SCG fast IRC clock configuration.

Public Members

uint32_t enableMode
Enable mode.

scg_firc_range_t range
Fast IRC frequency range.

const *scg_firc_trim_config_t* *trimConfig
Pointer to the FIRC trim configuration; set NULL to disable trim.

struct __fro192m__rf__clk__config
#include <fsl_clock.h> FRO192M RF clock configuration.

Public Members

fro192m_rf_range_t range
FRO192M RF clock frequency range.

fro192m_rf_clk_div_t apb__rfcmc__div
RF Flash APB and RF_CMC clock divide.

2.4 CMC: Core Mode Controller Driver

void CMC__SetClockMode(CMC_Type *base, *cmc_clock_mode_t* mode)
Sets clock mode.

This function configs the amount of clock gating when the core asserts Sleeping due to WFI, WFE or SLEEPONEXIT.

Parameters

- base – CMC peripheral base address.
- mode – System clock mode.

static inline void CMC__LockClockModeSetting(CMC_Type *base)
Locks the clock mode setting.

After invoking this function, any clock mode setting will be blocked.

Parameters

- base – CMC peripheral base address.

static inline *cmc_core_clock_gate_status_t* CMC__GetCoreClockGatedStatus(CMC_Type *base)
Gets the core clock gated status.

This function get the status to indicate whether the core clock is gated. The core clock gated status can be cleared by software.

Parameters

- base – CMC peripheral base address.

Returns
The status to indicate whether the core clock is gated.

static inline void CMC_ClearCoreClockGatedStatus(CMC_Type *base)

Clears the core clock gated status.

This function clear clock status flag by software.

Parameters

- base – CMC peripheral base address.

static inline uint8_t CMC_GetWakeupSource(CMC_Type *base)

Gets the Wakeup Source.

This function gets the Wakeup sources from the previous low power mode entry.

Parameters

- base – CMC peripheral base address.

Returns
The Wakeup sources from the previous low power mode entry. See _cmc_wakeup_sources for details.

static inline *cmc_clock_mode_t* CMC_GetClockMode(CMC_Type *base)

Gets the Clock mode.

This function gets the clock mode of the previous low power mode entry.

Parameters

- base – CMC peripheral base address.

Returns
The Low Power status.

static inline uint32_t CMC_GetSystemResetStatus(CMC_Type *base)

Gets the System reset status.

This function returns the system reset status. Those status updates on every MAIN Warm Reset to indicate the type/source of the most recent reset.

Parameters

- base – CMC peripheral base address.

Returns
The most recent system reset status. See _cmc_system_reset_sources for details.

static inline uint32_t CMC_GetStickySystemResetStatus(CMC_Type *base)

Gets the sticky system reset status since the last WAKE Cold Reset.

This function gets all source of system reset that have generated a system reset since the last WAKE Cold Reset, and that have not been cleared by software.

Parameters

- base – CMC peripheral base address.

Returns
System reset status that have not been cleared by software. See _cmc_system_reset_sources for details.

static inline void CMC_ClearStickySystemResetStatus(CMC_Type *base, uint32_t mask)

Clears the sticky system reset status flags.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the sticky system reset status to be cleared.

static inline uint8_t CMC_GetResetCount(CMC_Type *base)

Gets the number of reset sequences completed since the last WAKE Cold Reset.

Parameters

- base – CMC peripheral base address.

Returns

The number of reset sequences.

void CMC_SetPowerModeProtection(CMC_Type *base, uint32_t allowedModes)

Configures all power mode protection settings.

This function configures the power mode protection settings for supported power modes. This should be done before set the lowPower mode for each power doamin.

The allowed lowpower modes are passed as bit map. For example, to allow Sleep and DeepSleep, use CMC_SetPowerModeProtection(CMC_base, kCMC_AllowSleepMode|kCMC_AllowDeepSleepMode). To allow all low power modes, use CMC_SetPowerModeProtection(CMC_base, kCMC_AllowAllLowPowerModes).

Parameters

- base – CMC peripheral base address.
- allowedModes – Bitmaps of the allowed power modes. See _cmc_power_mode_protection for details.

static inline void CMC_LockPowerModeProtectionSetting(CMC_Type *base)

Locks the power mode protection.

This function locks the power mode protection. After invoking this function, any power mode protection setting will be ignored.

Parameters

- base – CMC peripheral base address.

static inline void CMC_SetGlobalPowerMode(CMC_Type *base, *cmc_low_power_mode_t* lowPowerMode)

Config the same lowPower mode for all power domain.

This function configures the same low power mode for MAIN power domian and WAKE power domain.

Parameters

- base – CMC peripheral base address.
- lowPowerMode – The desired lowPower mode. See cmc_low_power_mode_t for details.

static inline void CMC_SetMAINPowerMode(CMC_Type *base, *cmc_low_power_mode_t* lowPowerMode)

Configures entry into low power mode for the MAIN Power domain.

This function configures the low power mode for the MAIN power domian, when the core executes WFI/WFE instruction. The available lowPower modes are defined in the cmc_low_power_mode_t.

Parameters

- base – CMC peripheral base address.

- lowPowerMode – The desired lowPower mode. See cmc_low_power_mode_t for details.

static inline *cmc_low_power_mode_t* CMC_GetMAINPowerMode(CMC_Type *base)

Gets the power mode of the MAIN Power domain.

Parameters

- base – CMC peripheral base address.

Returns

The power mode of MAIN Power domain. See cmc_low_power_mode_t for details.

static inline void CMC_SetWAKEPowerMode(CMC_Type *base, *cmc_low_power_mode_t* lowPowerMode)

Configure entry into low power mode for the WAKE Power domain.

This function configures the low power mode for the WAKE power domian, when the core executes WFI/WFE instruction. The available lowPower mode are defined in the cmc_low_power_mode_t.

Note: The lowPower Mode for the WAKE domain must not be configured to a lower power mode than any other power domain.

Parameters

- base – CMC peripheral base address.
- lowPowerMode – The desired lowPower mode. See cmc_low_power_mode_t for details.

static inline *cmc_low_power_mode_t* CMC_GetWAKEPowerMode(CMC_Type *base)

Gets the power mode of the WAKE Power domain.

Parameters

- base – CMC peripheral base address.

Returns

The power mode of WAKE Power domain. See cmc_low_power_mode_t for details.

void CMC_ConfigResetPin(CMC_Type *base, const *cmc_reset_pin_config_t* *config)

Configure reset pin.

This function configures reset pin. When enabled, the low power filter is enabled in both Active and Low power modes, the reset filter is only enabled in Active mode. When both filers are enabled, they operate in series.

Parameters

- base – CMC peripheral base address.
- config – Pointer to the reset pin config structure.

static inline void CMC_EnableSystemResetInterrupt(CMC_Type *base, uint32_t mask)

Enable system reset interrupts.

This function enables the system reset interrupts. The assertion of non-fatal warm reset can be delayed for 258 cycles of the 32K_CLK clock while an enabled interrupt is generated. Then Software can perform a graceful shutdown or abort the non-fatal warm reset provided the pending reset source is cleared by resetting the reset source and then clearing the pending flag.

Parameters

- base – CMC peripheral base address.
- mask – System reset interrupts. See _cmc_system_reset_interrupt_enable for details.

static inline void CMC_DisableSystemResetInterrupt(CMC_Type *base, uint32_t mask)

Disable system reset interrupts.

This function disables the system reset interrupts.

Parameters

- base – CMC peripheral base address.
- mask – System reset interrupts. See _cmc_system_reset_interrupt_enable for details.

static inline uint32_t CMC_GetSystemResetInterruptFlags(CMC_Type *base)

Gets System Reset interrupt flags.

This function returns the System reset interrupt flags.

Parameters

- base – CMC peripheral base address.

Returns

System reset interrupt flags. See _cmc_system_reset_interrupt_flag for details.

static inline void CMC_ClearSystemResetInterruptFlags(CMC_Type *base, uint32_t mask)

Clears System Reset interrupt flags.

This function clears system reset interrupt flags. The pending reset source can be cleared by resetting the source of the reset and then clearing the pending flags.

Parameters

- base – CMC peripheral base address.
- mask – System Reset interrupt flags. See _cmc_system_reset_interrupt_flag for details.

static inline void CMC_EnableNonMaskablePinInterrupt(CMC_Type *base, bool enable)

Enable/Disable Non maskable Pin interrupt.

Parameters

- base – CMC peripheral base address.
- enable – Enable or disable Non maskable pin interrupt. true - enable Non-maskable pin interrupt. false - disable Non-maskable pin interupt.

static inline uint8_t CMC_GetISPMODEPinLogic(CMC_Type *base)

Gets the logic state of the ISPMODE_n pin.

This function returns the logic state of the ISPMODE_n pin on the last negation of RESET_b pin.

Parameters

- base – CMC peripheral base address.

Returns

The logic state of the ISPMODE_n pin on the last negation of RESET_b pin.

static inline void CMC_ClearISPMODEPinLogic(CMC_Type *base)

Clears ISPMODE_n pin state.

Parameters

- base – CMC peripheral base address.

static inline void CMC_ForceBootConfiguration(CMC_Type *base, bool assert)

Set the logic state of the BOOT_CONFIGn pin.

This function force the logic state of the Boot_Confign pin to assert on next system reset.

Parameters

- base – CMC peripheral base address.
- assert – Assert the corresponding pin or not. true - Assert corresponding pin on next system reset. false - No effect.

static inline void CMC_LockWriteOperationToBootRomStatusReg(CMC_Type *base, uint8_t index)

Lock write operation to BootROM status register and BootROM Lock register.

Note: If locked, BootROM status register cannot be written.

Note: Once locked, only cold reset can reset related register.

Parameters

- base – CMC peripheral base address.
- index – The index of BootROM status register, ranges from 0.

static inline bool CMC_CheckBootRomStatusRegWriteLocked(CMC_Type *base, uint8_t index)

Check if BootROM status register can be written.

Parameters

- base – CMC peripheral base address.
- index – The index of BootROM status register, ranges from 0.

Return values

- true – The selected BootRom status register is locked and cannot be written.
- false – The selected BootRom Status register is unlocked and cannot be written.

static inline uint32_t CMC_GetBootRomStatus(CMC_Type *base, uint8_t index)

Gets the information written by the BootROM.

Parameters

- base – CMC peripheral base address.
- index – The index of BootROM status register, ranges from 0.

Returns

The status information written by the BootROM.

static inline void CMC_WriteBootRomStatusReg(CMC_Type *base, uint8_t index, uint32_t value)

Writes value to BootROM status register, in this way, BootROM status registers are used as general purpose register.

Note: Value in BootROM status registers are reset in cold reset.

Parameters

- base – CMC peripheral base address.
- index – The index of BootROM status register, ranges from 0.
- value – Value to write.

void CMC_PowerOffSRAMAllMode(CMC_Type *base, uint32_t mask)

Power off the selected system SRAM always.

This function power off the selected system SRAM always. The SRAM arrays should not be accessed while they are shut down. SRAM array contents are not retained if they are powered off.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the SRAM arrays to be powered off all modes. See _cmc_system_sram_arrays for details. Check Reference Manual for the SRAM region and mask bit relationship.

static inline void CMC_PowerOnSRAMAllMode(CMC_Type *base, uint32_t mask)

Power on SRAM during all mode.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the SRAM arrays to be powered on all modes. See _cmc_system_sram_arrays for details. Check Reference Manual for the SRAM region and mask bit relationship.

void CMC_PowerOffSRAMLowPowerOnly(CMC_Type *base, uint32_t mask)

Power off the selected system SRAM during low power mode only.

This function power off the selected system SRAM only during low power mode. SRAM array contents are not retained if they are power off.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the SRAM arrays to be power off during low power mode only. See _cmc_system_sram_arrays for details. Check Reference Manual for the SRAM region and mask bit relationship.

static inline void CMC_PowerOnSRAMLowPowerOnly(CMC_Type *base, uint32_t mask)

Power on the selected system SRAM during low power mode only.

This function power on the selected system SRAM. The SRAM arrray contents are retained in low power modes.

Parameters

- base – CMC peripheral base address.
- mask – Bitmap of the SRAM arrays to be power on during low power mode only. See _cmc_system_sram_arrays for details. Check Reference Manual for the SRAM region and mask bit relationship.

void CMC_ConfigFlashMode(CMC_Type *base, bool wake, bool doze, bool disable)

Configs the low power mode of the on-chip flash memory.

This function configs the low power mode of the on-chip flash memory.

Parameters

- base – CMC peripheral base address.
- wake – true: Flash will exit low power state during the flash memory accesses. false: No effect.
- doze – true: Flash is disabled while core is sleeping false: No effect.
- disable – true: Flash memory is placed in low power state. false: No effect.

static inline void CMC_EnableDebugOperation(CMC_Type *base, bool enable)

Enables/Disables debug Operation when the core sleep.

This function configs what happens to debug when core sleeps.

Parameters

- base – CMC peripheral base address.
- enable – Enable or disable Debug when Core is sleeping. true - Debug remains enabled when the core is sleeping. false - Debug is disabled when the core is sleeping.

void CMC_PreEnterLowPowerMode(void)

Prepares to enter low power modes.

This function should be called before entering low power modes.

void CMC_PostExitLowPowerMode(void)

Recovers after wake up from low power modes.

This function should be called after wake up from low power modes. This function should be used with CMC_PreEnterLowPowerMode()

void CMC_GlobalEnterLowPowerMode(CMC_Type *base, *cmc_low_power_mode_t* lowPowerMode)

Configs the entry into the same low power mode for each power domains.

This function provides the feature to entry into the same low power mode for each power domains. Before invoking this function, please ensure the selected power mode have been allowed.

Parameters

- base – CMC peripheral base address.
- lowPowerMode – The low power mode to be entered. See cmc_low_power_mode_t for the details.

void CMC_EnterLowPowerMode(CMC_Type *base, const *cmc_power_domain_config_t* *config)

Configs the entry into different low power modes for each power domains.

This function provides the feature to entry into different low power modes for each power domains. Before invoking this function please ensure the selected modes are allowed.

Parameters

- base – CMC peripheral base address.
- config – Pointer to the cmc_power_domain_config_t structure.

FSL_CMC_DRIVER_VERSION

CMC driver version 2.4.4.

CMC_SRAM_BUSY_TIMEOUT

Max loops to wait for CMC SRAM operation complete.

When configuring the SRAM, driver will wait for the completion of new settings. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

enum _cmc_power_mode_protection

CMC power mode Protection enumeration.

Values:

enumerator kCMC_AllowSleepMode

Allow Sleep mode.

enumerator kCMC_AllowDeepSleepMode

Allow Deep Sleep mode.

enumerator kCMC_AllowPowerDownMode

Allow Power Down mode.

enumerator kCMC_AllowDeepPowerDownMode

Allow Deep Power Down mode.

enumerator kCMC_AllowAllLowPowerModes

Allow all low power modes.

enum _cmc_wakeup_sources

Wake up sources from the previous low power mode entry.

Values:

enumerator kCMC_WakeupFromResetInterruptOrPowerDown

Wakeup source is reset interrupt, or wake up from [Deep] Power Down.

enumerator kCMC_WakeupFromDebugRequest

Wakeup source is debug request.

enumerator kCMC_WakeupFromInterrupt

Wakeup source is interrupt.

enumerator kCMC_WakeupFromDMAWakeup

Wakeup source is DMA Wakeup.

enumerator kCMC_WakeupFromWUURequest

Wakeup source is WUU request.

enumerator kCMC_WakeupFromBusMaster

Wakeup source is Bus master.

enum _cmc_system_reset_interrupt_enable

System Reset Interrupt enable enumeration.

Values:

enumerator kCMC_PinResetInterruptEnable

Pin Reset interrupt enable.

enumerator kCMC_DAPResetInterruptEnable

DAP Reset interrupt enable.

enumerator kCMC_LowPowerAcknowledgeTimeoutResetInterruptEnable

Low Power Acknowledge Timeout Reset interrupt enable.

enumerator kCMC_Watchdog0ResetInterruptEnable
Watchdog 0 Reset interrupt enable.

enumerator kCMC_SoftwareResetInterruptEnable
Software Reset interrupt enable.

enumerator kCMC_LockupResetInterruptEnable
Lockup Reset interrupt enable.

enumerator kCMC_Watchdog1ResetInterruptEnable
Watchdog 1 Reset interrupt enable

enum _cmc_system_reset_interrupt_flag
CMC System Reset Interrupt Status flag.

Values:

enumerator kCMC_PinResetInterruptFlag
Pin Reset interrupt flag.

enumerator kCMC_DAPResetInterruptFlag
DAP Reset interrupt flag.

enumerator kCMC_LowPowerAcknowledgeTimeoutResetFlag
Low Power Acknowledge Timeout Reset interrupt flag.

enumerator kCMC_Watchdog0ResetInterruptFlag
Watchdog 0 Reset interrupt flag.

enumerator kCMC_SoftwareResetInterruptFlag
Software Reset interrupt flag.

enumerator kCMC_LockupResetInterruptFlag
Lock up Reset interrupt flag.

enumerator kCMC_Watchdog1ResetInterruptFlag
Watchdog 1 Reset interrupt flag.

enum _cmc_system_sram_arrays
CMC System SRAM arrays low power mode enable enumeration.

Values:

enumerator kCMC_SRAMBank0
Power off SRAM Bank0, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank1
Power off SRAM Bank1, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank2
Power off SRAM Bank2, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank3
Power off SRAM Bank3, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank4
Power off SRAM Bank4, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank5
Power off SRAM Bank5, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank6
Power off SRAM Bank6, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank7
Power off SRAM Bank7, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank8
Power off SRAM Bank8, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank9
Power off SRAM Bank9, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_SRAMBank10
Power off SRAM Bank10, please refer to chip's RM for the corresponding SRAM array.

enumerator kCMC_AllSramArrays
Mask of all system SRAM arrays.

enum _cmc_system_reset_sources
System reset sources enumeration.

Values:

enumerator kCMC_WakeUpReset
The reset caused by a wakeup from Power Down or Deep Power Down mode.

enumerator kCMC_PORReset
The reset caused by power on reset detection logic.

enumerator kCMC_LVDReset
The reset caused by a Low Voltage Detect.

enumerator kCMC_HVDReset
The reset caused by a High voltage Detect.

enumerator kCMC_WarmReset
The last reset source is a warm reset source.

enumerator kCMC_FatalReset
The last reset source is a fatal reset source.

enumerator kCMC_PinReset
The reset caused by the RESET_b pin.

enumerator kCMC_DAPReset
The reset caused by a reset request from the Debug Access port.

enumerator kCMC_ResetTimeout
The reset caused by a timeout or other error condition in the system reset generation.

enumerator kCMC_LowPowerAcknowledgeTimeoutReset
The reset caused by a timeout in low power mode entry logic.

enumerator kCMC_SCGReset
The reset caused by a loss of clock or loss of lock event in the SCG.

enumerator kCMC_Watchdog0Reset
The reset caused by a WatchDog 0 timeout.

enumerator kCMC_SoftwareReset
The reset caused by a software reset request.

enumerator kCMC_LockUpReset
The reset caused by the ARM core indication of a LOCKUP event.

enumerator kCMC_Watchdog1Reset
The reset caused by a WatchDog 1 timeout.

enum _cmc_core_clock_gate_status
Indicate the core clock was gated.

Values:

enumerator kCMC_CoreClockNotGated
Core clock not gated.

enumerator kCMC_CoreClockGated
Core clock was gated due to low power mode entry.

enum _cmc_clock_mode
CMC clock mode enumeration.

Values:

enumerator kCMC_GateNoneClock
No clock gating.

enumerator kCMC_GateCoreClock
Gate Core clock.

enumerator kCMC_GateCorePlatformClock
Gate Core clock and platform clock.

enumerator kCMC_GateAllSystemClocks
Gate all System clocks, without getting core entering into low power mode.

enumerator kCMC_GateAllSystemClocksEnterLowPowerMode
Gate all System clocks, with core entering into low power mode.

enum _cmc_low_power_mode
CMC power mode enumeration.

Values:

enumerator kCMC_ActiveMode
Select Active mode.

enumerator kCMC_SleepMode
Select Sleep mode when a core executes WFI or WFE instruction.

enumerator kCMC_DeepSleepMode
Select Deep Sleep mode when a core executes WFI or WFE instruction.

enumerator kCMC_PowerDownMode
Select Power Down mode when a core executes WFI or WFE instruction.

enumerator kCMC_DeepPowerDown
Select Deep Power Down mode when a core executes WFI or WFE instruction.

typedef enum *_cmc_core_clock_gate_status* cmc_core_clock_gate_status_t
Indicate the core clock was gated.

typedef enum *_cmc_clock_mode* cmc_clock_mode_t
CMC clock mode enumeration.

typedef enum *_cmc_low_power_mode* cmc_low_power_mode_t
CMC power mode enumeration.

typedef struct *_cmc_reset_pin_config* cmc_reset_pin_config_t
CMC reset pin configuration.

typedef struct *_cmc_power_domain_config* cmc_power_domain_config_t
power mode configuration for each power domain.

CMC_BLR_LOCK_FIELD_WIDTH

CMC_BLR_LOCK_IDX_MASK(index)

CMC_BLR_LOCK_IDX_SHIFT(index)

CMC_BLR_LOCK_IDX(index, value)

struct _cmc_reset_pin_config
#include <fsl_cmc.h> CMC reset pin configuration.

Public Members

bool lowpowerFilterEnable
Low Power Filter enable.

bool resetFilterEnable
Reset Filter enable.

uint8_t resetFilterWidth
Width of the Reset Filter.

struct _cmc_power_domain_config
#include <fsl_cmc.h> power mode configuration for each power domain.

Public Members

cmc_clock_mode_t clock_mode
Clock mode for each power domain.

cmc_low_power_mode_t main_domain
The low power mode of the MAIN power domain.

cmc_low_power_mode_t wake_domain
The low power mode of the WAKE power domain.

2.5 CRC: Cyclic Redundancy Check Driver

FSL_CRC_DRIVER_VERSION
CRC driver version. Version 2.0.5.

Current version: 2.0.5

Change log:

- Version 2.0.5
  - Fix CERT-C issue with boolean-to-unsigned integer conversion.
- Version 2.0.4
  - Release peripheral from reset if necessary in init function.

- Version 2.0.3
  - Fix MISRA issues
- Version 2.0.2
  - Fix MISRA issues
- Version 2.0.1
  - move DATA and DATALL macro definition from header file to source file

enum _crc_bits

CRC bit width.

Values:

enumerator kCrcBits16

Generate 16-bit CRC code

enumerator kCrcBits32

Generate 32-bit CRC code

enum _crc_result

CRC result type.

Values:

enumerator kCrcFinalChecksum

CRC data register read value is the final checksum. Reflect out and final xor protocol features are applied.

enumerator kCrcIntermediateChecksum

CRC data register read value is intermediate checksum (raw value). Reflect out and final xor protocol feature are not applied. Intermediate checksum can be used as a seed for CRC_Init() to continue adding data to this checksum.

typedef enum *_crc_bits* crc_bits_t

CRC bit width.

typedef enum *_crc_result* crc_result_t

CRC result type.

typedef struct *_crc_config* crc_config_t

CRC protocol configuration.

This structure holds the configuration for the CRC protocol.

void CRC_Init(CRC_Type *base, const *crc_config_t* *config)

Enables and configures the CRC peripheral module.

This function enables the clock gate in the SIM module for the CRC peripheral. It also configures the CRC module and starts a checksum computation by writing the seed.

Parameters

- base – CRC peripheral address.
- config – CRC module configuration structure.

static inline void CRC_Deinit(CRC_Type *base)

Disables the CRC peripheral module.

This function disables the clock gate in the SIM module for the CRC peripheral.

Parameters

- base – CRC peripheral address.

void CRC_GetDefaultConfig(*crc_config_t* *config)

Loads default values to the CRC protocol configuration structure.

Loads default values to the CRC protocol configuration structure. The default values are as follows.

```
config->polynomial = 0x1021;
config->seed = 0xFFFF;
config->reflectIn = false;
config->reflectOut = false;
config->complementChecksum = false;
config->crcBits = kCrcBits16;
config->crcResult = kCrcFinalChecksum;
```

Parameters

- config – CRC protocol configuration structure.

void CRC_WriteData(CRC_Type *base, const uint8_t *data, size_t dataSize)

Writes data to the CRC module.

Writes input data buffer bytes to the CRC data register. The configured type of transpose is applied.

Parameters

- base – CRC peripheral address.
- data – Input data stream, MSByte in data[0].
- dataSize – Size in bytes of the input data buffer.

uint32_t CRC_Get32bitResult(CRC_Type *base)

Reads the 32-bit checksum from the CRC module.

Reads the CRC data register (either an intermediate or the final checksum). The configured type of transpose and complement is applied.

Parameters

- base – CRC peripheral address.

Returns

An intermediate or the final 32-bit checksum, after configured transpose and complement operations.

uint16_t CRC_Get16bitResult(CRC_Type *base)

Reads a 16-bit checksum from the CRC module.

Reads the CRC data register (either an intermediate or the final checksum). The configured type of transpose and complement is applied.

Parameters

- base – CRC peripheral address.

Returns

An intermediate or the final 16-bit checksum, after configured transpose and complement operations.

CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT

Default configuration structure filled by CRC_GetDefaultConfig(). Use CRC16-CCIT-FALSE as defeault.

struct _crc_config

#include <fsl_crc.h> CRC protocol configuration.

This structure holds the configuration for the CRC protocol.

Public Members

uint32_t polynomial
CRC Polynomial, MSBit first. Example polynomial: 0x1021 = 1_0000_0010_0001 = x^12+x^5+1

uint32_t seed
Starting checksum value

bool reflectIn
Reflect bits on input.

bool reflectOut
Reflect bits on output.

bool complementChecksum
True if the result shall be complement of the actual checksum.

crc_bits_t crcBits
Selects 16- or 32- bit CRC protocol.

crc_result_t crcResult
Selects final or intermediate checksum return from CRC_Get16bitResult() or CRC_Get32bitResult()

2.6 EDMA: Enhanced Direct Memory Access (eDMA) Controller Driver

void EDMA_Init(DMA_Type *base, const *edma_config_t* *config)
Initializes the eDMA peripheral.

This function ungates the eDMA clock and configures the eDMA peripheral according to the configuration structure.

Note: This function enables the minor loop map feature.

Parameters

- base – eDMA peripheral base address.
- config – A pointer to the configuration structure, see "edma_config_t".

void EDMA_Deinit(DMA_Type *base)
Deinitializes the eDMA peripheral.

This function gates the eDMA clock.

Parameters

- base – eDMA peripheral base address.

void EDMA_InstallTCD(DMA_Type *base, uint32_t channel, *edma_tcd_t* *tcd)
Push content of TCD structure into hardware TCD register.

Parameters

- base – EDMA peripheral base address.
- channel – EDMA channel number.
- tcd – Point to TCD structure.

void EDMA_GetDefaultConfig(*edma_config_t* *config)

Gets the eDMA default configuration structure.

This function sets the configuration structure to default values. The default configuration is set to the following values:

```
config.enableMasterIdReplication = true;
config.enableHaltOnError = true;
config.enableRoundRobinArbitration = false;
config.enableDebugMode = false;
config.enableBufferedWrites = false;
```

Parameters

- config – A pointer to the eDMA configuration structure.

static inline void EDMA_EnableAllChannelLink(DMA_Type *base, bool enable)

Enables/disables all channel linking.

This function enables/disables all channel linking in the management page. For specific channel linking enablement & configuration, please refer to EDMA_SetChannelLink and EDMA_TcdSetChannelLink APIs.

For example, to disable all channel linking in the DMA0 management page:

```
EDMA_EnableAllChannelLink(DMA0, false);
```

Parameters

- base – eDMA peripheral base address.
- enable – Switcher of the channel linking feature for all channels. "true" means to enable. "false" means not.

void EDMA_ResetChannel(DMA_Type *base, uint32_t channel)

Sets all TCD registers to default values.

This function sets TCD registers for this channel to default values.

Note: This function must not be called while the channel transfer is ongoing or it causes unpredictable results.

Note: This function enables the auto stop request feature.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.

void EDMA_SetTransferConfig(DMA_Type *base, uint32_t channel, const *edma_transfer_config_t* *config, *edma_tcd_t* *nextTcd)

Configures the eDMA transfer attribute.

This function configures the transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the TCD address. Example:

```
edma_transfer_config_t config;
edma_tcd_t tcd;
config.srcAddr = ..;
config.destAddr = ..;
...
EDMA_SetTransferConfig(DMA0, channel, &config, &stcd);
```

Note: If nextTcd is not NULL, it means scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the eDMA_ResetChannel.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- config – Pointer to eDMA transfer configuration structure.
- nextTcd – Point to TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

void EDMA_SetMinorOffsetConfig(DMA_Type *base, uint32_t channel, const *edma_minor_offset_config_t* *config)

Configures the eDMA minor offset feature.

The minor offset means that the signed-extended value is added to the source address or destination address after each minor loop.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- config – A pointer to the minor offset configuration structure.

static inline void EDMA_SetChannelArbitrationGroup(DMA_Type *base, uint32_t channel, uint32_t group)

Configures the eDMA channel arbitration group.

This function configures the channel arbitration group. The arbitration group priorities are evaluated by numeric value from highest group number to lowest.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number
- group – Fixed-priority arbitration group number for the channel.

static inline void EDMA_SetChannelPreemptionConfig(DMA_Type *base, uint32_t channel, const *edma_channel_Preemption_config_t* *config)

Configures the eDMA channel preemption feature.

This function configures the channel preemption attribute and the priority of the channel.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number
- config – A pointer to the channel preemption configuration structure.

static inline uint32_t EDMA_GetChannelSystemBusInformation(DMA_Type *base, uint32_t channel)

Gets the eDMA channel identification and attribute information on the system bus interface.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.

Returns

The mask of the channel system bus information. Users need to use the _edma_channel_sys_bus_info type to decode the return variables.

void EDMA_SetChannelLink(DMA_Type *base, uint32_t channel, *edma_channel_link_type_t* type, uint32_t linkedChannel)

Sets the channel link for the eDMA transfer.

This function configures either the minor link or the major link mode. The minor link means that the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note: Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- type – A channel link type, which can be one of the following:
  - kEDMA_LinkNone
  - kEDMA_MinorLink
  - kEDMA_MajorLink
- linkedChannel – The linked channel number.

void EDMA_SetBandWidth(DMA_Type *base, uint32_t channel, *edma_bandwidth_t* bandWidth)

Sets the bandwidth for the eDMA transfer.

Because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. The bandwidth forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- bandWidth – A bandwidth setting, which can be one of the following:
  - kEDMABandwidthStallNone
  - kEDMABandwidthStall4Cycle
  - kEDMABandwidthStall8Cycle

void EDMA_SetModulo(DMA_Type *base, uint32_t channel, *edma_modulo_t* srcModulo, *edma_modulo_t* destModulo)

Sets the source modulo and the destination modulo for the eDMA transfer.

This function defines a specific address range specified to be the value after (SADDR + SOFF)/(DADDR + DOFF) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- srcModulo – A source modulo value.
- destModulo – A destination modulo value.

static inline void EDMA_EnableAsyncRequest(DMA_Type *base, uint32_t channel, bool enable)

Enables an async request for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- enable – The command to enable (true) or disable (false).

static inline void EDMA_EnableAutoStopRequest(DMA_Type *base, uint32_t channel, bool enable)

Enables an auto stop request for the eDMA transfer.

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- enable – The command to enable (true) or disable (false).

void EDMA_EnableChannelInterrupts(DMA_Type *base, uint32_t channel, uint32_t mask)

Enables the interrupt source for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mask – The mask of interrupt source to be set. Users need to use the defined edma_interrupt_enable_t type.

void EDMA_DisableChannelInterrupts(DMA_Type *base, uint32_t channel, uint32_t mask)

Disables the interrupt source for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mask – The mask of the interrupt source to be set. Use the defined edma_interrupt_enable_t type.

static inline void EDMA_SetChannelMux(DMA_Type *base, uint32_t channel, uint32_t mux)

Set channel mux source.

Note:When the peripheral is no longer needed, the mux configuration for that channel should be written to 0, thus releasing the resource.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mux – the mux source value is SOC specific, please reference the SOC for detail.

void EDMA_TcdReset(*edma_tcd_t* *tcd)

Sets all fields to default values for the TCD structure.

This function sets all fields for this TCD structure to default value.

Note: This function enables the auto stop request feature.

Parameters

- tcd – Pointer to the TCD structure.

void EDMA_TcdSetTransferConfig(*edma_tcd_t* *tcd, const *edma_transfer_config_t* *config, *edma_tcd_t* *nextTcd)

Configures the eDMA TCD transfer attribute.

The TCD is a transfer control descriptor. The content of the TCD is the same as the hardware TCD registers. The STCD is used in the scatter-gather mode. This function configures the TCD transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the next TCD address. Example:

```
edma_transfer_config_t config = {
...
}
edma_tcd_t tcd __aligned(32);
edma_tcd_t nextTcd __aligned(32);
EDMA_TcdSetTransferConfig(&tcd, &config, &nextTcd);
```

Note: TCD address should be 32 bytes aligned or it causes an eDMA error.

Note: If the nextTcd is not NULL, the scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the EDMA_TcdReset.

Parameters

- tcd – Pointer to the TCD structure.
- config – Pointer to eDMA transfer configuration structure.
- nextTcd – Pointer to the next TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

void EDMA_TcdSetMinorOffsetConfig(*edma_tcd_t* *tcd, const *edma_minor_offset_config_t* *config)

Configures the eDMA TCD minor offset feature.

A minor offset is a signed-extended value added to the source address or a destination address after each minor loop.

Parameters

- tcd – A point to the TCD structure.
- config – A pointer to the minor offset configuration structure.

void EDMA_TcdSetChannelLink(*edma_tcd_t* *tcd, *edma_channel_link_type_t* type, uint32_t linkedChannel)

Sets the channel link for the eDMA TCD.

This function configures either a minor link or a major link. The minor link means the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note: Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

- tcd – Point to the TCD structure.
- type – Channel link type, it can be one of:
  - kEDMA_LinkNone
  - kEDMA_MinorLink
  - kEDMA_MajorLink
- linkedChannel – The linked channel number.

static inline void EDMA_TcdSetBandWidth(*edma_tcd_t* *tcd, *edma_bandwidth_t* bandWidth)

Sets the bandwidth for the eDMA TCD.

Because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. The bandwidth forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Parameters

- tcd – A pointer to the TCD structure.
- bandWidth – A bandwidth setting, which can be one of the following:
  - kEDMABandwidthStallNone
  - kEDMABandwidthStall4Cycle
  - kEDMABandwidthStall8Cycle

void EDMA_TcdSetModulo(*edma_tcd_t* *tcd, *edma_modulo_t* srcModulo, *edma_modulo_t* destModulo)

Sets the source modulo and the destination modulo for the eDMA TCD.

This function defines a specific address range specified to be the value after (SADDR + SOFF)/(DADDR + DOFF) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

- tcd – A pointer to the TCD structure.
- srcModulo – A source modulo value.
- destModulo – A destination modulo value.

static inline void EDMA_TcdEnableAutoStopRequest(*edma_tcd_t* *tcd, bool enable)

Sets the auto stop request for the eDMA TCD.

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

- tcd – A pointer to the TCD structure.
- enable – The command to enable (true) or disable (false).

void EDMA_TcdEnableInterrupts(*edma_tcd_t* *tcd, uint32_t mask)

Enables the interrupt source for the eDMA TCD.

Parameters

- tcd – Point to the TCD structure.
- mask – The mask of interrupt source to be set. Users need to use the defined edma_interrupt_enable_t type.

void EDMA_TcdDisableInterrupts(*edma_tcd_t* *tcd, uint32_t mask)

Disables the interrupt source for the eDMA TCD.

Parameters

- tcd – Point to the TCD structure.
- mask – The mask of interrupt source to be set. Users need to use the defined edma_interrupt_enable_t type.

static inline void EDMA_EnableChannelRequest(DMA_Type *base, uint32_t channel)

Enables the eDMA hardware channel request.

This function enables the hardware channel request.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.

static inline void EDMA_DisableChannelRequest(DMA_Type *base, uint32_t channel)

Disables the eDMA hardware channel request.

This function disables the hardware channel request.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.

static inline void EDMA_TriggerChannelStart(DMA_Type *base, uint32_t channel)

Starts the eDMA transfer by using the software trigger.

This function starts a minor loop transfer.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.

uint32_t EDMA_GetRemainingMajorLoopCount(DMA_Type *base, uint32_t channel)

Gets the Remaining major loop count from the eDMA current channel TCD.

This function checks the TCD (Task Control Descriptor) status for a specified eDMA channel and returns the number of major loop count that has not finished.

Note: 1. This function can only be used to get unfinished major loop count of transfer without the next TCD, or it might be inaccuracy.

a. The unfinished/remaining transfer bytes cannot be obtained directly from registers while the channel is running. Because to calculate the remaining bytes, the initial NBYTES configured in DMA_TCDn_NBYTES_MLNO register is needed while the eDMA IP does not support getting it while a channel is active. In another word, the NBYTES value reading is always the actual (decrementing) NBYTES value the dma_engine is working with while a channel is running. Consequently, to get the remaining transfer bytes, a software-saved initial value of NBYTES (for example copied before enabling the channel) is needed. The formula to calculate it is shown below: RemainingBytes = RemainingMajorLoopCount * NBYTES(initially configured)

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.

Returns

Major loop count which has not been transferred yet for the current TCD.

static inline uint32_t EDMA_GetErrorStatusFlags(DMA_Type *base)

Gets the eDMA channel error status flags.

Parameters

- base – eDMA peripheral base address.

Returns

The mask of error status flags. Users need to use the _edma_error_status_flags type to decode the return variables.

uint32_t EDMA_GetChannelStatusFlags(DMA_Type *base, uint32_t channel)

Gets the eDMA channel status flags.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.

Returns

The mask of channel status flags. Users need to use the _edma_channel_status_flags type to decode the return variables.

void EDMA_ClearChannelStatusFlags(DMA_Type *base, uint32_t channel, uint32_t mask)

Clears the eDMA channel status flags.

Parameters

- base – eDMA peripheral base address.
- channel – eDMA channel number.
- mask – The mask of channel status to be cleared. Users need to use the defined _edma_channel_status_flags type.

void EDMA_CreateHandle(*edma_handle_t* *handle, DMA_Type *base, uint32_t channel)

Creates the eDMA handle.

This function is called if using the transactional API for eDMA. This function initializes the internal state of the eDMA handle.

Parameters

- handle – eDMA handle pointer. The eDMA handle stores callback function and parameters.
- base – eDMA peripheral base address.
- channel – eDMA channel number.

void EDMA_InstallTCDMemory(*edma_handle_t* *handle, *edma_tcd_t* *tcdPool, uint32_t tcdSize)

Installs the TCDs memory pool into the eDMA handle.

This function is called after the EDMA_CreateHandle to use scatter/gather feature.

Parameters

- handle – eDMA handle pointer.
- tcdPool – A memory pool to store TCDs. It must be 32 bytes aligned.
- tcdSize – The number of TCD slots.

void EDMA_SetCallback(*edma_handle_t* *handle, *edma_callback* callback, void *userData)

Installs a callback function for the eDMA transfer.

This callback is called in the eDMA IRQ handler. Use the callback to do something after the current major loop transfer completes.

Parameters

- handle – eDMA handle pointer.
- callback – eDMA callback function pointer.
- userData – A parameter for the callback function.

void EDMA_PrepareTransferConfig(*edma_transfer_config_t* *config, void *srcAddr, uint32_t srcWidth, int16_t srcOffset, void *destAddr, uint32_t destWidth, int16_t destOffset, uint32_t bytesEachRequest, uint32_t transferBytes)

Prepares the eDMA transfer structure configurations.

This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

Parameters

- config – The user configuration structure of type edma_transfer_config_t.
- srcAddr – eDMA transfer source address.
- srcWidth – eDMA transfer source address width(bytes).
- srcOffset – eDMA transfer source address offset
- destAddr – eDMA transfer destination address.
- destWidth – eDMA transfer destination address width(bytes).
- destOffset – eDMA transfer destination address offset

- bytesEachRequest – eDMA transfer bytes per channel request.
- transferBytes – eDMA transfer bytes to be transferred.

void EDMA_PrepareTransfer(*edma_transfer_config_t* *config, void *srcAddr, uint32_t srcWidth, void *destAddr, uint32_t destWidth, uint32_t bytesEachRequest, uint32_t transferBytes, *edma_transfer_type_t* transferType)

Prepares the eDMA transfer structure.

This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

Parameters

- config – The user configuration structure of type edma_transfer_config_t.
- srcAddr – eDMA transfer source address.
- srcWidth – eDMA transfer source address width(bytes).
- destAddr – eDMA transfer destination address.
- destWidth – eDMA transfer destination address width(bytes).
- bytesEachRequest – eDMA transfer bytes per channel request.
- transferBytes – eDMA transfer bytes to be transferred.
- transferType – eDMA transfer type.

status_t EDMA_SubmitTransfer(*edma_handle_t* *handle, const *edma_transfer_config_t* *config)

Submits the eDMA transfer request.

This function submits the eDMA transfer request according to the transfer configuration structure. If submitting the transfer request repeatedly, this function packs an unprocessed request as a TCD and enables scatter/gather feature to process it in the next time.

Parameters

- handle – eDMA handle pointer.
- config – Pointer to eDMA transfer configuration structure.

Return values

- kStatus_EDMA_Success – It means submit transfer request succeed.
- kStatus_EDMA_QueueFull – It means TCD queue is full. Submit transfer request is not allowed.
- kStatus_EDMA_Busy – It means the given channel is busy, need to submit request later.

void EDMA_StartTransfer(*edma_handle_t* *handle)

eDMA starts transfer.

This function enables the channel request. Users can call this function after submitting the transfer request or before submitting the transfer request.

Parameters

- handle – eDMA handle pointer.

void EDMA_StopTransfer(*edma_handle_t* *handle)

eDMA stops transfer.

This function disables the channel request to pause the transfer. Users can call EDMA_StartTransfer() again to resume the transfer.

Parameters

- handle – eDMA handle pointer.

void EDMA_AbortTransfer(*edma_handle_t* *handle)

eDMA aborts transfer.

This function disables the channel request and clear transfer status bits. Users can submit another transfer after calling this API.

Parameters

- handle – DMA handle pointer.

static inline uint32_t EDMA_GetUnusedTCDNumber(*edma_handle_t* *handle)

Get unused TCD slot number.

This function gets current tcd index which is run. If the TCD pool pointer is NULL, it will return 0.

Parameters

- handle – DMA handle pointer.

Returns

The unused tcd slot number.

static inline uint32_t EDMA_GetNextTCDAddress(*edma_handle_t* *handle)

Get the next tcd address.

This function gets the next tcd address. If this is last TCD, return 0.

Parameters

- handle – DMA handle pointer.

Returns

The next TCD address.

static inline *edma_transfer_size_t* EDMA_GetTransferSize(uint32_t width)

Get the transfer size.

This function gets the transfer size.

Parameters

- width – transfer width(bytes).

Returns

The transfer size.

void EDMA_HandleIRQ(*edma_handle_t* *handle)

eDMA IRQ handler for the current major loop transfer completion.

This function clears the channel major interrupt flag and calls the callback function if it is not NULL.

Parameters

- handle – eDMA handle pointer.

FSL_EDMA_DRIVER_VERSION

eDMA driver version

Version 2.5.3.

enum _edma_transfer_size

eDMA transfer configuration

Values:

enumerator kEDMA_TransferSize1Bytes

Source/Destination data transfer size is 1 byte every time

enumerator kEDMA_TransferSize2Bytes

Source/Destination data transfer size is 2 bytes every time

enumerator kEDMA_TransferSize4Bytes

Source/Destination data transfer size is 4 bytes every time

enumerator kEDMA_TransferSize8Bytes

Source/Destination data transfer size is 8 bytes every time

enumerator kEDMA_TransferSize16Bytes

Source/Destination data transfer size is 16 bytes every time

enumerator kEDMA_TransferSize32Bytes

Source/Destination data transfer size is 32 bytes every time

enumerator kEDMA_TransferSize64Bytes

Source/Destination data transfer size is 64 bytes every time

enum _edma_modulo

eDMA modulo configuration

Values:

enumerator kEDMA_ModuloDisable

Disable modulo

enumerator kEDMA_Modulo2bytes

Circular buffer size is 2 bytes.

enumerator kEDMA_Modulo4bytes

Circular buffer size is 4 bytes.

enumerator kEDMA_Modulo8bytes

Circular buffer size is 8 bytes.

enumerator kEDMA_Modulo16bytes

Circular buffer size is 16 bytes.

enumerator kEDMA_Modulo32bytes

Circular buffer size is 32 bytes.

enumerator kEDMA_Modulo64bytes

Circular buffer size is 64 bytes.

enumerator kEDMA_Modulo128bytes

Circular buffer size is 128 bytes.

enumerator kEDMA_Modulo256bytes

Circular buffer size is 256 bytes.

enumerator kEDMA_Modulo512bytes
Circular buffer size is 512 bytes.

enumerator kEDMA_Modulo1Kbytes
Circular buffer size is 1 K bytes.

enumerator kEDMA_Modulo2Kbytes
Circular buffer size is 2 K bytes.

enumerator kEDMA_Modulo4Kbytes
Circular buffer size is 4 K bytes.

enumerator kEDMA_Modulo8Kbytes
Circular buffer size is 8 K bytes.

enumerator kEDMA_Modulo16Kbytes
Circular buffer size is 16 K bytes.

enumerator kEDMA_Modulo32Kbytes
Circular buffer size is 32 K bytes.

enumerator kEDMA_Modulo64Kbytes
Circular buffer size is 64 K bytes.

enumerator kEDMA_Modulo128Kbytes
Circular buffer size is 128 K bytes.

enumerator kEDMA_Modulo256Kbytes
Circular buffer size is 256 K bytes.

enumerator kEDMA_Modulo512Kbytes
Circular buffer size is 512 K bytes.

enumerator kEDMA_Modulo1Mbytes
Circular buffer size is 1 M bytes.

enumerator kEDMA_Modulo2Mbytes
Circular buffer size is 2 M bytes.

enumerator kEDMA_Modulo4Mbytes
Circular buffer size is 4 M bytes.

enumerator kEDMA_Modulo8Mbytes
Circular buffer size is 8 M bytes.

enumerator kEDMA_Modulo16Mbytes
Circular buffer size is 16 M bytes.

enumerator kEDMA_Modulo32Mbytes
Circular buffer size is 32 M bytes.

enumerator kEDMA_Modulo64Mbytes
Circular buffer size is 64 M bytes.

enumerator kEDMA_Modulo128Mbytes
Circular buffer size is 128 M bytes.

enumerator kEDMA_Modulo256Mbytes
Circular buffer size is 256 M bytes.

enumerator kEDMA_Modulo512Mbytes
Circular buffer size is 512 M bytes.

enumerator kEDMA_Modulo1Gbytes
Circular buffer size is 1 G bytes.

enumerator kEDMA_Modulo2Gbytes
Circular buffer size is 2 G bytes.

enum _edma_bandwidth
Bandwidth control.

Values:

enumerator kEDMA_BandwidthStallNone
No eDMA engine stalls.

enumerator kEDMA_BandwidthStall4Cycle
eDMA engine stalls for 4 cycles after each read/write.

enumerator kEDMA_BandwidthStall8Cycle
eDMA engine stalls for 8 cycles after each read/write.

enum _edma_channel_link_type
Channel link type.

Values:

enumerator kEDMA_LinkNone
No channel link

enumerator kEDMA_MinorLink
Channel link after each minor loop

enumerator kEDMA_MajorLink
Channel link while major loop count exhausted

eDMA channel status flags, _edma_channel_status_flags

Values:

enumerator kEDMA_DoneFlag
DONE flag, set while transfer finished, CITER value exhausted

enumerator kEDMA_ErrorFlag
eDMA error flag, an error occurred in a transfer

enumerator kEDMA_InterruptFlag
eDMA interrupt flag, set while an interrupt occurred of this channel

eDMA channel error status flags, _edma_error_status_flags

Values:

enumerator kEDMA_DestinationBusErrorFlag
Bus error on destination address

enumerator kEDMA_SourceBusErrorFlag
Bus error on the source address

enumerator kEDMA_ScatterGatherErrorFlag
Error on the Scatter/Gather address, not 32byte aligned.

enumerator kEDMA_NbytesErrorFlag
NBYTES/CITER configuration error

enumerator kEDMA_DestinationOffsetErrorFlag
Destination offset not aligned with destination size

enumerator kEDMA_DestinationAddressErrorFlag
Destination address not aligned with destination size

enumerator kEDMA_SourceOffsetErrorFlag
Source offset not aligned with source size

enumerator kEDMA_SourceAddressErrorFlag
Source address not aligned with source size

enumerator kEDMA_TransferCanceledFlag
Transfer cancelled

enumerator kEDMA_ErrorChannelFlag
Error channel number of the cancelled channel number

enumerator kEDMA_ValidFlag
No error occurred, this bit is 0. Otherwise, it is 1.

eDMA channel system bus information, _edma_channel_sys_bus_info

Values:

enumerator kEDMA_AttributeOutput
DMA's AHB system bus attribute output value.

enumerator kEDMA_PrivilegedAccessLevel
Privileged Access Level for DMA transfers. 0b - User protection level; 1b - Privileged protection level.

enumerator kEDMA_MasterId
DMA's master ID when channel is active and master ID replication is enabled.

enum _edma_interrupt_enable
eDMA interrupt source

Values:

enumerator kEDMA_ErrorInterruptEnable
Enable interrupt while channel error occurs.

enumerator kEDMA_MajorInterruptEnable
Enable interrupt while major count exhausted.

enumerator kEDMA_HalfInterruptEnable
Enable interrupt while major count to half value.

enum _edma_transfer_type
eDMA transfer type

Values:

enumerator kEDMA_MemoryToMemory
Transfer from memory to memory

enumerator kEDMA_PeripheralToMemory
Transfer from peripheral to memory

enumerator kEDMA_MemoryToPeripheral
Transfer from memory to peripheral

enumerator kEDMA_PeripheralToPeripheral
Transfer from Peripheral to peripheral

eDMA transfer status, _edma_transfer_status

Values:

enumerator kStatus_EDMA_QueueFull
TCD queue is full.

enumerator kStatus_EDMA_Busy
Channel is busy and can't handle the transfer request.

typedef enum *_edma_transfer_size* edma_transfer_size_t
eDMA transfer configuration

typedef enum *_edma_modulo* edma_modulo_t
eDMA modulo configuration

typedef enum *_edma_bandwidth* edma_bandwidth_t
Bandwidth control.

typedef enum *_edma_channel_link_type* edma_channel_link_type_t
Channel link type.

typedef enum *_edma_interrupt_enable* edma_interrupt_enable_t
eDMA interrupt source

typedef enum *_edma_transfer_type* edma_transfer_type_t
eDMA transfer type

typedef struct *_edma_config* edma_config_t
eDMA global configuration structure.

typedef struct *_edma_transfer_config* edma_transfer_config_t
eDMA transfer configuration

This structure configures the source/destination transfer attribute.

typedef struct *_edma_channel_Preemption_config* edma_channel_Preemption_config_t
eDMA channel priority configuration

typedef struct *_edma_minor_offset_config* edma_minor_offset_config_t
eDMA minor offset configuration

typedef struct *_edma_tcd* edma_tcd_t
eDMA TCD.

This structure is same as TCD register which is described in reference manual, and is used to configure the scatter/gather feature as a next hardware TCD.

typedef void (*edma_callback)(struct *_edma_handle* *handle, void *userData, bool transferDone, uint32_t tcds)
Define callback function for eDMA.

typedef uint32_t (*edma_memorymap_callback)(uint32_t addr)
Memroy map function callback for DMA.

typedef struct *_edma_handle* edma_handle_t
eDMA transfer handle structure

struct _edma_config
#include <fsl_edma.h> eDMA global configuration structure.

Public Members

bool enableMasterIdReplication

Enable (true) master ID replication. If Master ID replication is disabled, the privileged protection level (supervisor mode) for DMA transfers is used.

bool enableHaltOnError

Enable (true) transfer halt on error. Any error causes the HALT bit to set. Subsequently, all service requests are ignored until the HALT bit is cleared.

bool enableRoundRobinArbitration

Enable (true) round robin channel arbitration method or fixed priority arbitration is used for channel selection

bool enableDebugMode

Enable(true) eDMA debug mode. When in debug mode, the eDMA stalls the start of a new channel. Executing channels are allowed to complete.

struct __edma__transfer__config

#include <fsl_edma.h> eDMA transfer configuration

This structure configures the source/destination transfer attribute.

Public Members

uint32_t srcAddr

Source data address.

uint32_t destAddr

Destination data address.

edma_transfer_size_t srcTransferSize

Source data transfer size.

edma_transfer_size_t destTransferSize

Destination data transfer size.

int16_t srcOffset

Sign-extended offset applied to the current source address to form the next-state value as each source read is completed.

int16_t destOffset

Sign-extended offset applied to the current destination address to form the next-state value as each destination write is completed.

uint32_t minorLoopBytes

Bytes to transfer in a minor loop

uint32_t majorLoopCounts

Major loop iteration count.

struct __edma__channel__Preemption__config

#include <fsl_edma.h> eDMA channel priority configuration

Public Members

bool enableChannelPreemption

If true: a channel can be suspended by other channel with higher priority

bool enablePreemptAbility
If true: a channel can suspend other channel with low priority

uint8_t channelPriority
Channel priority

struct __edma_minor_offset_config
#include <fsl_edma.h> eDMA minor offset configuration

Public Members

bool enableSrcMinorOffset
Enable(true) or Disable(false) source minor loop offset.

bool enableDestMinorOffset
Enable(true) or Disable(false) destination minor loop offset.

uint32_t minorOffset
Offset for a minor loop mapping.

struct __edma_tcd
#include <fsl_edma.h> eDMA TCD.

This structure is same as TCD register which is described in reference manual, and is used to configure the scatter/gather feature as a next hardware TCD.

Public Members

__IO uint32_t SADDR
SADDR register, used to save source address

__IO uint16_t SOFF
SOFF register, save offset bytes every transfer

__IO uint16_t ATTR
ATTR register, source/destination transfer size and modulo

__IO uint32_t NBYTES
Nbytes register, minor loop length in bytes

__IO uint32_t SLAST
SLAST register

__IO uint32_t DADDR
DADDR register, used for destination address

__IO uint16_t DOFF
DOFF register, used for destination offset

__IO uint16_t CITER
CITER register, current minor loop numbers, for unfinished minor loop.

__IO uint32_t DLAST_SGA
DLASTSGA register, next stcd address used in scatter-gather mode

__IO uint16_t CSR
CSR register, for TCD control status

__IO uint16_t BITER
BITER register, begin minor loop count.

struct _edma_handle

#include <fsl_edma.h> eDMA transfer handle structure

Public Members

edma_callback callback

Callback function for major count exhausted.

void *userData

Callback function parameter.

DMA_Type *base

eDMA peripheral base address.

edma_tcd_t *tcdPool

Pointer to memory stored TCDs.

uint8_t channel

eDMA channel number.

volatile int8_t header

The first TCD index.

volatile int8_t tail

The last TCD index.

volatile int8_t tcdUsed

The number of used TCD slots.

volatile int8_t tcdSize

The total number of TCD slots in the queue.

uint8_t flags

The status of the current channel.

2.7 ELEMU: Edgelock Messaging unit driver

Values:

enumerator kStatus_ELEMU_AgumentOutOfRange

ELEMU status for out of range access.

enumerator kStatus_ELEMU_InvalidArgument

ELEMU status for invalid argument check.

enumerator kStatus_ELEMU_RequestTimeout

ELEMU status for timeout.

enumerator kStatus_ELEMU_Busy

ELEMU status for reservation by other core.

enum elemu_ownership_status_t

Values:

enumerator kStatus_ELEMU_Unknown

Unexpected ELEMU ownership.

enumerator kStatus_ELEMU_LockedByMe
ELEMU reserved for current core.

enumerator kStatus_ELEMU_LockedByOther
ELEMU reserved for other core.

enumerator kStatus_ELEMU_Free
ELEMU not reserved.

typedef struct *mu_hdr* mu_hdr_t

void ELEMU_mu_hal_send_data(ELEMU_Type *mu, uint8_t regid, uint32_t *data)
Send one word of data to specified transmit register.

This function waits until the transmit register is ready and writes one word of data.

Parameters

- mu – ELEMU peripheral base address
- regid – Transmit register index
- data – Pointer to data to be sent

void ELEMU_mu_hal_receive_data(ELEMU_Type *mu, uint8_t regid, uint32_t *data)
Receive one word of data from specified receive register.

This function waits until data is available in the receive register and reads it.

Parameters

- mu – ELEMU peripheral base address
- regid – Receive register index
- data – Pointer to store received data

status_t ELEMU_mu_hal_receive_data_wait(ELEMU_Type *mu, uint8_t regid, uint32_t *data, uint32_t wait)
Wait for data availability and receive one word from specified register.

Parameters

- mu – ELEMU peripheral base address
- regid – Receive register index
- data – Pointer to store received data
- wait – Maximum wait iterations

Returns
kStatus_Success if successful, kStatus_ELEMU_RequestTimeout if timeout

status_t ELEMU_mu_read_message(ELEMU_Type *mu, uint32_t *buf, uint8_t *size, uint8_t read_header)
Read a message from ELEMU receive registers.

This function reads a message from ELEMU including header and payload.

Parameters

- mu – ELEMU peripheral base address
- buf – Pointer to buffer to store message
- size – Pointer to size variable
- read_header – Flag to indicate if header should be read

Returns
kStatus_Success if successful, kStatus_ELEMU_InvalidArgument if buf or size is NULL

status_t ELEMU_mu_read_data_wait(ELEMU_Type *mu, uint32_t buf[], uint8_t *size, uint32_t wait)

Wait and read data from ELEMU receive registers.

This function waits for data availability and reads the specified number of words.

Parameters

- mu – ELEMU peripheral base address
- buf – Pointer to buffer to store data
- size – Pointer to size variable
- wait – Maximum wait iterations

Returns
kStatus_Success if successful, kStatus_ELEMU_InvalidArgument if buf or size is NULL, kStatus_ELEMU_RequestTimeout if timeout

status_t ELEMU_mu_send_message(ELEMU_Type *mu, uint32_t buf[], size_t wordCount)

Send a message to ELEMU using transmit registers.

This function sends a message buffer to ELEMU by writing words to transmit registers.

Parameters

- mu – ELEMU peripheral base address
- buf – Pointer to message buffer
- wordCount – Number of words to send

Returns
kStatus_Success if successful, kStatus_ELEMU_InvalidArgument if buf is NULL, kStatus_ELEMU_AgumentOutOfRange if wordCount exceeds register count

status_t ELEMU_mu_get_response(ELEMU_Type *mu, uint32_t *buf, size_t wordCount)

Get response message from ELEMU.

This function reads a response message from ELEMU using the read_message function.

Parameters

- mu – ELEMU peripheral base address
- buf – Pointer to buffer to store response
- wordCount – Expected number of words

Returns
kStatus_Success if successful, kStatus_ELEMU_InvalidArgument if buf is NULL, kStatus_ELEMU_AgumentOutOfRange if wordCount exceeds register count

status_t ELEMU_mu_wait_for_data(ELEMU_Type *mu, uint32_t *buf, size_t wordCount, uint32_t wait)

Wait for data availability and read from ELEMU.

This function waits for data and reads the specified number of words.

Parameters

- mu – ELEMU peripheral base address

- buf – Pointer to buffer to store data
- wordCount – Expected number of words
- wait – Maximum wait iterations

Returns
kStatus_Success if successful, kStatus_ELEMU_InvalidArgument if buf is NULL, kStatus_ELEMU_AgumentOutOfRange if wordCount exceeds register count

status_t ELEMU_mu_wait_for_ready(ELEMU_Type *mu, uint32_t wait)

Wait until ELEMU is ready for communication.

This function checks the status register and waits until ELEMU is ready.

Parameters

- mu – ELEMU peripheral base address
- wait – Maximum wait iterations

Returns
kStatus_Success if ready, kStatus_ELEMU_RequestTimeout if timeout

elemu_ownership_status_t ELEMU_mu_get_ownership_status(ELEMU_Type *mu)

Get current ownership status of ELEMU.

Parameters

- mu – ELEMU peripheral base address

Returns
Ownership status enum value

status_t ELEMU_mu_get_ownership(ELEMU_Type *mu)

Attempt to acquire ownership of ELEMU.

This function tries to acquire ownership of ELEMU using the semaphore register.

Parameters

- mu – ELEMU peripheral base address

Returns
kStatus_Success if acquired, kStatus_ELEMU_Busy if locked by other, kStatus_Fail if acquisition failed

status_t ELEMU_mu_release_ownership(ELEMU_Type *mu)

Release ownership of ELEMU.

This function releases ownership of ELEMU using the semaphore release register.

Parameters

- mu – ELEMU peripheral base address

Returns
kStatus_Success if released, kStatus_ELEMU_Busy if locked by other, kStatus_Fail if release failed

status_t ELEMU_mu_release_ownership_force(ELEMU_Type *mu)

Release ownership of ELEMU.

This function releases ownership of ELEMU using the semaphore release register.

Parameters

- mu – ELEMU peripheral base address

Returns
kStatus_Success if released, kStatus_ELEMU_Busy if locked by other, kStatus_Fail if release failed

void ELEMU_mu_init(ELEMU_Type *mu)

Initialize ELEMU module.

This function does nothing. ELEMU is initialized after leaving ROM.

Parameters

- mu – ELEMU peripheral base address

status_t ELEMU_LP_WakeupPathInit(ELEMU_Type *mu)

Initialize low-power wakeup path for ELEMU.

This function enables clocks and waits for ELEMU startup sequence.

Parameters

- mu – ELEMU peripheral base address

Returns
kStatus_Success if successful, kStatus_Fail otherwise

struct mu_hdr

#include <fsl_elemu.h>

2.8 EWM: External Watchdog Monitor Driver

void EWM_Init(EWM_Type *base, const *ewm_config_t* *config)

Initializes the EWM peripheral.

This function is used to initialize the EWM. After calling, the EWM runs immediately according to the configuration. Note that, except for the interrupt enable control bit, other control bits and registers are write once after a CPU reset. Modifying them more than once generates a bus transfer error.

This is an example.

```
ewm_config_t config;
EWM_GetDefaultConfig(&config);
config.compareHighValue = 0xAAU;
EWM_Init(ewm_base,&config);
```

Parameters

- base – EWM peripheral base address
- config – The configuration of the EWM

void EWM_Deinit(EWM_Type *base)

Deinitializes the EWM peripheral.

This function is used to shut down the EWM.

Parameters

- base – EWM peripheral base address

void EWM_GetDefaultConfig(*ewm_config_t* *config)

Initializes the EWM configuration structure.

This function initializes the EWM configuration structure to default values. The default values are as follows.

```
ewmConfig->enableEwm = true;
ewmConfig->enableEwmInput = false;
ewmConfig->setInputAssertLogic = false;
ewmConfig->enableInterrupt = false;
ewmConfig->ewm_lpo_clock_source_t = kEWM_LpoClockSource0;
ewmConfig->prescaler = 0;
ewmConfig->compareLowValue = 0;
ewmConfig->compareHighValue = 0xFEU;
```

See also:

ewm_config_t

Parameters

- config – Pointer to the EWM configuration structure.

static inline void EWM_EnableInterrupts(EWM_Type *base, uint32_t mask)

Enables the EWM interrupt.

This function enables the EWM interrupt.

Parameters

- base – EWM peripheral base address
- mask – The interrupts to enable The parameter can be combination of the following source if defined
  - kEWM_InterruptEnable

static inline void EWM_DisableInterrupts(EWM_Type *base, uint32_t mask)

Disables the EWM interrupt.

This function enables the EWM interrupt.

Parameters

- base – EWM peripheral base address
- mask – The interrupts to disable The parameter can be combination of the following source if defined
  - kEWM_InterruptEnable

static inline uint32_t EWM_GetStatusFlags(EWM_Type *base)

Gets all status flags.

This function gets all status flags.

This is an example for getting the running flag.

```
uint32_t status;
status = EWM_GetStatusFlags(ewm_base) & kEWM_RunningFlag;
```

See also:

_ewm_status_flags_t

- True: a related status flag has been set.
- False: a related status flag is not set.

Parameters

- base – EWM peripheral base address

Returns
State of the status flag: asserted (true) or not-asserted (false).

void EWM_Refresh(EWM_Type *base)

Services the EWM.

This function resets the EWM counter to zero.

Parameters

- base – EWM peripheral base address

FSL_EWM_DRIVER_VERSION

EWM driver version 2.0.4.

enum _ewm_lpo_clock_source

Describes EWM clock source.

Values:

enumerator kEWM_LpoClockSource0
EWM clock sourced from lpo_clk[0]

enumerator kEWM_LpoClockSource1
EWM clock sourced from lpo_clk[1]

enumerator kEWM_LpoClockSource2
EWM clock sourced from lpo_clk[2]

enumerator kEWM_LpoClockSource3
EWM clock sourced from lpo_clk[3]

enum _ewm_interrupt_enable_t

EWM interrupt configuration structure with default settings all disabled.

This structure contains the settings for all of EWM interrupt configurations.

Values:

enumerator kEWM_InterruptEnable
Enable the EWM to generate an interrupt

enum _ewm_status_flags_t

EWM status flags.

This structure contains the constants for the EWM status flags for use in the EWM functions.

Values:

enumerator kEWM_RunningFlag
Running flag, set when EWM is enabled

typedef enum *_ewm_lpo_clock_source* ewm_lpo_clock_source_t

Describes EWM clock source.

typedef struct *_ewm_config* ewm_config_t

Data structure for EWM configuration.

This structure is used to configure the EWM.

struct _ewm_config

#include <fsl_ewm.h> Data structure for EWM configuration.

This structure is used to configure the EWM.

Public Members

bool enableEwm
Enable EWM module

bool enableEwmInput
Enable EWM_in input

bool setInputAssertLogic
EWM_in signal assertion state

bool enableInterrupt
Enable EWM interrupt

ewm_lpo_clock_source_t clockSource
Clock source select

uint8_t prescaler
Clock prescaler value

uint8_t compareLowValue
Compare low-register value

uint8_t compareHighValue
Compare high-register value

2.9 FGPIO Driver

2.10 C40ESP3 Flash Driver

enum _flash_driver_api_keys
Enumeration for Flash driver API keys.

Note: The resulting value is built with a byte order such that the string being readable in expected order when viewed in a hex editor, if the value is treated as a 32-bit little endian value.

Values:

enumerator kFLASH_ApiEraseKey
Key value used to validate all flash erase APIs.

status_t FLASH_Init(*flash_config_t* *config)
Initializes the global flash properties structure members.

This function checks and initializes the Flash module for the other Flash APIs.

Parameters

- config – Pointer to the storage for the driver runtime state.

Return values

- kStatus_FLASH_Success – API was executed successfully.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.

- kStatus_FLASH_CommandNotSupported – Flash API is not supported.

status_t FLASH_Erase(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes, uint32_t key)

Erases the flash sectors encompassed by parameters passed into function.

status_t FLASH_EraseAll(FMU_Type *base, uint32_t key)

Erases entire flash and ifr.

status_t FLASH_Program(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint8_t *src, uint32_t lengthInBytes)

Programs flash phrases with data at locations passed in through parameters.

status_t FLASH_ProgramPage(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint8_t *src, uint32_t lengthInBytes)

Programs flash pages with data at locations passed in through parameters.

status_t FLASH_VerifyErasePhrase(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes)

Verify that the flash phrases are erased.

status_t FLASH_VerifyErasePage(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes)

Verify that the flash pages are erased.

status_t FLASH_VerifyEraseSector(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes)

Verify that the flash sectors are erased.

status_t FLASH_VerifyEraseAll(FMU_Type *base)

Verify that all flash and IFR space is erased.

status_t FLASH_VerifyEraseBlock(*flash_config_t* *config, FMU_Type *base, uint32_t blockaddr)

Verify that a flash block is erased.

status_t FLASH_VerifyEraseIFRPhrase(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes)

Verify that the ifr phrases are erased.

status_t FLASH_VerifyEraseIFRPage(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes)

Verify that the ifr pages are erased.

status_t FLASH_VerifyEraseIFRSector(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t lengthInBytes)

Verify that the ifr sectors are erased.

status_t FLASH_GetProperty(*flash_config_t* *config, *flash_property_tag_t* whichProperty, uint32_t *value)

Returns the desired flash property.

status_t Read_Into_MISR(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t ending, uint32_t *seed, uint32_t *signature)

Read into MISR.

The Read into MISR operation generates a signature based on the contents of the selected flash memory using an embedded MISR.

status_t Read_IFR_Into_MISR(*flash_config_t* *config, FMU_Type *base, uint32_t start, uint32_t ending, uint32_t *seed, uint32_t *signature)

Read IFR into MISR.

The Read IFR into MISR operation generates a signature based on the contents of the selected IFR space using an embedded MISR.

FSL_FLASH_DRIVER_VERSION

Flash driver version for SDK.

Version 2.3.3.

enum _flash_driver_version_constants

Flash driver version for ROM.

Values:

enumerator kFLASH_DriverVersionName

Flash driver version name.

enumerator kFLASH_DriverVersionMajor

Major flash driver version.

enumerator kFLASH_DriverVersionMinor

Minor flash driver version.

enumerator kFLASH_DriverVersionBugfix

Bugfix for flash driver version.

enum _flash_property_tag

Enumeration for various flash properties.

Values:

enumerator kFLASH_PropertyPflash0SectorSize

Pflash sector size property.

enumerator kFLASH_PropertyPflash0TotalSize

Pflash total size property.

enumerator kFLASH_PropertyPflash0BlockSize

Pflash block size property.

enumerator kFLASH_PropertyPflash0BlockCount

Pflash block count property.

enumerator kFLASH_PropertyPflash0BlockBaseAddr

Pflash block base address property.

enumerator kFLASH_PropertyPflash0FacSupport

Pflash fac support property.

enumerator kFLASH_PropertyPflash0AccessSegmentSize

Pflash access segment size property.

enumerator kFLASH_PropertyPflash0AccessSegmentCount

Pflash access segment count property.

enumerator kFLASH_PropertyPflash1SectorSize

Pflash sector size property.

enumerator kFLASH_PropertyPflash1TotalSize

Pflash total size property.

enumerator kFLASH_PropertyPflash1BlockSize
Pflash block size property.

enumerator kFLASH_PropertyPflash1BlockCount
Pflash block count property.

enumerator kFLASH_PropertyPflash1BlockBaseAddr
Pflash block base address property.

enumerator kFLASH_PropertyPflash1FacSupport
Pflash fac support property.

enumerator kFLASH_PropertyPflash1AccessSegmentSize
Pflash access segment size property.

enumerator kFLASH_PropertyPflash1AccessSegmentCount
Pflash access segment count property.

enumerator kFLASH_PropertyFlexRamBlockBaseAddr
FlexRam block base address property.

enumerator kFLASH_PropertyFlexRamTotalSize
FlexRam total size property.

typedef enum *_flash_property_tag* flash_property_tag_t
Enumeration for various flash properties.

typedef struct *_flash_mem_descriptor* flash_mem_desc_t
Flash memory descriptor.

typedef struct *_flash_ifr_desc* flash_ifr_desc_t

typedef struct *_msf1_config* msf1_config_t

typedef struct *_flash_config* flash_config_t
Flash driver state information.

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

FLASH_ADDR_MASK

struct _flash_mem_descriptor
#include <fsl_k4_flash.h> Flash memory descriptor.

Public Members

uint32_t blockBase
Base address of the flash block

uint32_t totalSize
The size of the flash block.

uint32_t blockCount
A number of flash blocks.

struct _flash_ifr_desc
#include <fsl_k4_flash.h>

struct _msf1_config
#include <fsl_k4_flash.h>

struct _flash_config

#include <fsl_k4_flash.h> Flash driver state information.

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

2.11 FlexCAN: Flex Controller Area Network Driver

2.12 FlexCAN Driver

bool FLEXCAN_IsInstanceHasFDMode(CAN_Type *base)

Determine whether the FlexCAN instance support CAN FD mode at run time.

Note: Use this API only if different soc parts share the SOC part name macro define. Otherwise, a different SOC part name can be used to determine at compile time whether the FlexCAN instance supports CAN FD mode or not. If need use this API to determine if CAN FD mode is supported, the FLEXCAN_Init function needs to be executed first, and then call this API and use the return to value determines whether to supports CAN FD mode, if return true, continue calling FLEXCAN_FDInit to enable CAN FD mode.

Parameters

- base – FlexCAN peripheral base address.

Returns

return TRUE if instance support CAN FD mode, FALSE if instance only support classic CAN (2.0) mode.

uint32_t FLEXCAN_GetFDMailboxOffset(CAN_Type *base, uint8_t mbIdx)

Get Mailbox offset number by dword.

This function gets the offset number of the specified mailbox. Mailbox is not consecutive between memory regions when payload is not 8 bytes so need to calculate the specified mailbox address. For example, in the first memory region, MB[0].CS address is 0x4002_4080. For 32 bytes payload frame, the second mailbox is ((1/12)*512 + 1%12*40)/4 = 10, meaning 10 dword after the 0x4002_4080, which is actually the address of mailbox MB[1].CS.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – Mailbox index.

Returns

Mailbox address offset in word.

status_t FLEXCAN_EnterFreezeMode(CAN_Type *base)

Enter FlexCAN Freeze Mode.

This function makes the FlexCAN work under Freeze Mode.

Parameters

- base – FlexCAN peripheral base address.

Returns

kStatus_Success Enter Freeze Mode successful kStatus_Timeout Timeout when wait for Freeze Mode Acknowledge

status_t FLEXCAN_ExitFreezeMode(CAN_Type *base)

Exit FlexCAN Freeze Mode.

This function makes the FlexCAN leave Freeze Mode.

Parameters

- base – FlexCAN peripheral base address.

Returns

kStatus_Success Enter Freeze Mode successful kStatus_Timeout Timeout when wait for Freeze Mode Acknowledge

uint32_t FLEXCAN_GetInstance(CAN_Type *base)

Get the FlexCAN instance from peripheral base address.

Parameters

- base – FlexCAN peripheral base address.

Returns

FlexCAN instance.

bool FLEXCAN_CalculateImprovedTimingValues(CAN_Type *base, uint32_t bitRate, uint32_t sourceClock_Hz, *flexcan_timing_config_t* *pTimingConfig)

Calculates the improved timing values by specific bit Rates for classical CAN.

This function use to calculates the Classical CAN timing values according to the given bit rate. The Calculated timing values will be set in CTRL1/CBT/ENCBT register. The calculation is based on the recommendation of the CiA 301 v4.2.0 and previous version document.

Parameters

- base – FlexCAN peripheral base address.
- bitRate – The classical CAN speed in bps defined by user, should be less than or equal to 1Mbps.
- sourceClock_Hz – The Source clock frequency in Hz.
- pTimingConfig – Pointer to the FlexCAN timing configuration structure.

Returns

TRUE if timing configuration found, FALSE if failed to find configuration.

void FLEXCAN_Init(CAN_Type *base, const *flexcan_config_t* *pConfig, uint32_t sourceClock_Hz)

Initializes a FlexCAN instance.

This function initializes the FlexCAN module with user-defined settings. This example shows how to set up the flexcan_config_t parameters and how to call the FLEXCAN_Init function by passing in these parameters.

```
flexcan_config_t flexcanConfig;
flexcanConfig.clkSrc                = kFLEXCAN_ClkSrc0;
flexcanConfig.bitRate               = 1000000U;
flexcanConfig.maxMbNum              = 16;
flexcanConfig.enableLoopBack        = false;
flexcanConfig.enableSelfWakeup      = false;
flexcanConfig.enableIndividMask     = false;
flexcanConfig.enableDoze            = false;
flexcanConfig.disableSelfReception = false;
flexcanConfig.enableListenOnlyMode = false;
flexcanConfig.timingConfig          = timingConfig;
FLEXCAN_Init(CAN0, &flexcanConfig, 40000000UL);
```

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the user-defined configuration structure.
- sourceClock_Hz – FlexCAN Protocol Engine clock source frequency in Hz.

bool FLEXCAN_FDCalculateImprovedTimingValues(CAN_Type *base, uint32_t bitRate, uint32_t bitRateFD, uint32_t sourceClock_Hz, *flexcan_timing_config_t* *pTimingConfig)

Calculates the improved timing values by specific bit rates for CANFD.

This function use to calculates the CANFD timing values according to the given nominal phase bit rate and data phase bit rate. The Calculated timing values will be set in CBT/ENCBT and FDCBT/EDCBT registers. The calculation is based on the recommendation of the CiA 1301 v1.0.0 document.

Parameters

- base – FlexCAN peripheral base address.
- bitRate – The CANFD bus control speed in bps defined by user.
- bitRateFD – The CAN FD data phase speed in bps defined by user. Equal to bitRate means disable bit rate switching.
- sourceClock_Hz – The Source clock frequency in Hz.
- pTimingConfig – Pointer to the FlexCAN timing configuration structure.

Returns

TRUE if timing configuration found, FALSE if failed to find configuration

void FLEXCAN_FDInit(CAN_Type *base, const *flexcan_config_t* *pConfig, uint32_t sourceClock_Hz, *flexcan_mb_size_t* dataSize, bool brs)

Initializes a FlexCAN instance.

This function initializes the FlexCAN module with user-defined settings. This example shows how to set up the flexcan_config_t parameters and how to call the FLEXCAN_FDInit function by passing in these parameters.

```
flexcan_config_t flexcanConfig;
flexcanConfig.clkSrc                = kFLEXCAN_ClkSrc0;
flexcanConfig.bitRate               = 1000000U;
flexcanConfig.bitRateFD             = 2000000U;
flexcanConfig.maxMbNum              = 16;
flexcanConfig.enableLoopBack        = false;
flexcanConfig.enableSelfWakeup      = false;
flexcanConfig.enableIndividMask     = false;
flexcanConfig.disableSelfReception = false;
flexcanConfig.enableListenOnlyMode = false;
flexcanConfig.enableDoze            = false;
flexcanConfig.timingConfig          = timingConfig;
FLEXCAN_FDInit(CAN0, &flexcanConfig, 80000000UL, kFLEXCAN_16BperMB, true);
```

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the user-defined configuration structure.
- sourceClock_Hz – FlexCAN Protocol Engine clock source frequency in Hz.
- dataSize – FlexCAN Message Buffer payload size. The actual transmitted or received CAN FD frame data size needs to be less than or equal to this value.
- brs – True if bit rate switch is enabled in FD mode.

void FLEXCAN_Deinit(CAN_Type *base)

De-initializes a FlexCAN instance.

This function disables the FlexCAN module clock and sets all register values to the reset value.

Parameters

- base – FlexCAN peripheral base address.

void FLEXCAN_GetDefaultConfig(*flexcan_config_t* *pConfig)

Gets the default configuration structure.

This function initializes the FlexCAN configuration structure to default values. The default values are as follows. flexcanConfig->clkSrc = kFLEXCAN_ClkSrc0; flexcanConfig->bitRate = 1000000U; flexcanConfig->bitRateFD = 2000000U; flexcanConfig->maxMbNum = 16; flexcanConfig->enableLoopBack = false; flexcanConfig->enableSelfWakeup = false; flexcanConfig->enableIndividMask = false; flexcanConfig->disableSelfReception = false; flexcanConfig->enableListenOnlyMode = false; flexcanConfig->enableDoze = false; flexcanConfig->enablePretendedeNetworking = false; flexcanConfig->enableMemoryErrorControl = true; flexcanConfig->enableNonCorrectableErrorEnterFreeze = true; flexcanConfig->enableTransceiverDelayMeasure = true; flexcanConfig->enableRemoteRequestFrameStored = true; flexcanConfig->payloadEndianness = kFLEXCAN_bigEndian; flexcanConfig.timingConfig = timingConfig;

Parameters

- pConfig – Pointer to the FlexCAN configuration structure.

void FLEXCAN_SetTimingConfig(CAN_Type *base, const *flexcan_timing_config_t* *pConfig)

Sets the FlexCAN classical CAN protocol timing characteristic.

This function gives user settings to classical CAN or CAN FD nominal phase timing characteristic. The function is for an experienced user. For less experienced users, call the FLEXCAN_SetBitRate() instead.

Note: Calling FLEXCAN_SetTimingConfig() overrides the bit rate set in FLEXCAN_Init() or FLEXCAN_SetBitRate().

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the timing configuration structure.

status_t FLEXCAN_SetBitRate(CAN_Type *base, uint32_t sourceClock_Hz, uint32_t bitRate_Bps)

Set bit rate of FlexCAN classical CAN frame or CAN FD frame nominal phase.

This function set the bit rate of classical CAN frame or CAN FD frame nominal phase base on FLEXCAN_CalculateImprovedTimingValues() API calculated timing values.

Note: Calling FLEXCAN_SetBitRate() overrides the bit rate set in FLEXCAN_Init().

Parameters

- base – FlexCAN peripheral base address.
- sourceClock_Hz – Source Clock in Hz.
- bitRate_Bps – Bit rate in Bps.

Returns
kStatus_Success - Set CAN baud rate (only Nominal phase) successfully.

void FLEXCAN_SetFDTimingConfig(CAN_Type *base, const *flexcan_timing_config_t* *pConfig)

Sets the FlexCAN CANFD data phase timing characteristic.

This function gives user settings to CANFD data phase timing characteristic. The function is for an experienced user. For less experienced users, call the FLEXCAN_SetFDBitRate() to set both Nominal/Data bit Rate instead.

Note: Calling FLEXCAN_SetFDTimingConfig() overrides the data phase bit rate set in FLEXCAN_FDInit()/FLEXCAN_SetFDBitRate().

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the timing configuration structure.

status_t FLEXCAN_SetFDBitRate(CAN_Type *base, uint32_t sourceClock_Hz, uint32_t bitRateN_Bps, uint32_t bitRateD_Bps)

Set bit rate of FlexCAN FD frame.

This function set the baud rate of FLEXCAN FD base on FLEXCAN_FDCalculateImprovedTimingValues() API calculated timing values.

Parameters

- base – FlexCAN peripheral base address.
- sourceClock_Hz – Source Clock in Hz.
- bitRateN_Bps – Nominal bit Rate in Bps.
- bitRateD_Bps – Data bit Rate in Bps.

Returns
kStatus_Success - Set CAN FD bit rate (include Nominal and Data phase) successfully.

void FLEXCAN_SetRxMbGlobalMask(CAN_Type *base, uint32_t mask)

Sets the FlexCAN receive message buffer global mask.

This function sets the global mask for the FlexCAN message buffer in a matching process. The configuration is only effective when the Rx individual mask is disabled in the FLEXCAN_Init().

Parameters

- base – FlexCAN peripheral base address.
- mask – Rx Message Buffer Global Mask value.

void FLEXCAN_SetRxFifoGlobalMask(CAN_Type *base, uint32_t mask)

Sets the FlexCAN receive FIFO global mask.

This function sets the global mask for FlexCAN FIFO in a matching process.

Parameters

- base – FlexCAN peripheral base address.
- mask – Rx Fifo Global Mask value.

void FLEXCAN_SetRxIndividualMask(CAN_Type *base, uint8_t maskIdx, uint32_t mask)

Sets the FlexCAN receive individual mask.

This function sets the individual mask for the FlexCAN matching process. The configuration is only effective when the Rx individual mask is enabled in the FLEXCAN_Init(). If the Rx FIFO is disabled, the individual mask is applied to the corresponding Message Buffer. If the Rx FIFO is enabled, the individual mask for Rx FIFO occupied Message Buffer is applied to the Rx Filter with the same index. Note that only the first 32 individual masks can be used as the Rx FIFO filter mask.

Parameters

- base – FlexCAN peripheral base address.
- maskIdx – The Index of individual Mask.
- mask – Rx Individual Mask value.

void FLEXCAN_SetTxMbConfig(CAN_Type *base, uint8_t mbIdx, bool enable)

Configures a FlexCAN transmit message buffer.

This function aborts the previous transmission, cleans the Message Buffer, and configures it as a Transmit Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- enable – Enable/disable Tx Message Buffer.
  - true: Enable Tx Message Buffer.
  - false: Disable Tx Message Buffer.

void FLEXCAN_SetRxMbConfig(CAN_Type *base, uint8_t mbIdx, const *flexcan_rx_mb_config_t* *pRxMbConfig, bool enable)

Configures a FlexCAN Receive Message Buffer.

This function cleans a FlexCAN build-in Message Buffer and configures it as a Receive Message Buffer. User should invoke this API when CTRL2[RRS]=1. When CTRL2[RRS]=1, frame's ID is compared to the IDs of the receive mailboxes with the CODE field configured as kFLEXCAN_RxMbEmpty, kFLEXCAN_RxMbFull or kFLEXCAN_RxMbOverrun. Message buffer will store the remote frame in the same fashion of a data frame. No automatic remote response frame will be generated. User need to setup another message buffer to respond remote request.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- pRxMbConfig – Pointer to the FlexCAN Message Buffer configuration structure.
- enable – Enable/disable Rx Message Buffer.
  - true: Enable Rx Message Buffer.
  - false: Disable Rx Message Buffer.

static inline void FLEXCAN_SetMbID(CAN_Type *base, uint8_t mbIdx, uint32_t id)

Configures a FlexCAN Message Buffer identifier.

Parameters

- base – FlexCAN peripheral base address.

- mbIdx – The Message Buffer index.
- id – CAN Message Buffer Identifier, should use FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.

void FLEXCAN_SetFDTxMbConfig(CAN_Type *base, uint8_t mbIdx, bool enable)

Configures a FlexCAN transmit message buffer.

This function aborts the previous transmission, cleans the Message Buffer, and configures it as a Transmit Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- enable – Enable/disable Tx Message Buffer.
  - true: Enable Tx Message Buffer.
  - false: Disable Tx Message Buffer.

void FLEXCAN_SetFDRxMbConfig(CAN_Type *base, uint8_t mbIdx, const *flexcan_rx_mb_config_t* *pRxMbConfig, bool enable)

Configures a FlexCAN Receive Message Buffer.

This function cleans a FlexCAN build-in Message Buffer and configures it as a Receive Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- pRxMbConfig – Pointer to the FlexCAN Message Buffer configuration structure.
- enable – Enable/disable Rx Message Buffer.
  - true: Enable Rx Message Buffer.
  - false: Disable Rx Message Buffer.

static inline void FLEXCAN_SetFDMbID(CAN_Type *base, uint8_t mbIdx, uint32_t id)

Configures a FlexCAN Message Buffer identifier.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The Message Buffer index.
- id – CAN Message Buffer Identifier, should use FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.

void FLEXCAN_SetRemoteResponseMbConfig(CAN_Type *base, uint8_t mbIdx, const *flexcan_frame_t* *pFrame)

Configures a FlexCAN Remote Response Message Buffer.

User should invoke this API when CTRL2[RRS]=0. When CTRL2[RRS]=0, frame's ID is compared to the IDs of the receive mailboxes with the CODE field configured as kFLEXCAN_RxMbRanswer. If there is a matching ID, then this mailbox content will be transmitted as response. The received remote request frame is not stored in receive buffer. It is only used to trigger a transmission of a frame in response.

Parameters

- base – FlexCAN peripheral base address.

- mbIdx – The Message Buffer index.
- pFrame – Pointer to CAN message frame structure for response.

void FLEXCAN_SetRxFifoConfig(CAN_Type *base, const *flexcan_rx_fifo_config_t* *pRxFifoConfig, bool enable)

Configures the FlexCAN Legacy Rx FIFO.

This function configures the FlexCAN Rx FIFO with given configuration.

Note: Legacy Rx FIFO only can receive classic CAN message.

Parameters

- base – FlexCAN peripheral base address.
- pRxFifoConfig – Pointer to the FlexCAN Legacy Rx FIFO configuration structure. Can be NULL when enable parameter is false.
- enable – Enable/disable Legacy Rx FIFO.
  - true: Enable Legacy Rx FIFO.
  - false: Disable Legacy Rx FIFO.

void FLEXCAN_SetEnhancedRxFifoConfig(CAN_Type *base, const *flexcan_enhanced_rx_fifo_config_t* *pConfig, bool enable)

Configures the FlexCAN Enhanced Rx FIFO.

This function configures the Enhanced Rx FIFO with given configuration.

Note: Enhanced Rx FIFO support receive classic CAN or CAN FD messages, Legacy Rx FIFO and Enhanced Rx FIFO cannot be enabled at the same time.

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the FlexCAN Enhanced Rx FIFO configuration structure. Can be NULL when enable parameter is false.
- enable – Enable/disable Enhanced Rx FIFO.
  - true: Enable Enhanced Rx FIFO.
  - false: Disable Enhanced Rx FIFO.

void FLEXCAN_SetPNConfig(CAN_Type *base, const *flexcan_pn_config_t* *pConfig)

Configures the FlexCAN Pretended Networking mode.

This function configures the FlexCAN Pretended Networking mode with given configuration.

Parameters

- base – FlexCAN peripheral base address.
- pConfig – Pointer to the FlexCAN Rx FIFO configuration structure.

static inline uint64_t FLEXCAN_GetStatusFlags(CAN_Type *base)

Gets the FlexCAN module interrupt flags.

This function gets all FlexCAN status flags. The flags are returned as the logical OR value of the enumerators _flexcan_flags. To check the specific status, compare the return value with enumerators in _flexcan_flags.

Parameters

- base – FlexCAN peripheral base address.

Returns

FlexCAN status flags which are ORed by the enumerators in the _flexcan_flags.

static inline void FLEXCAN_ClearStatusFlags(CAN_Type *base, uint64_t mask)

Clears status flags with the provided mask.

This function clears the FlexCAN status flags with a provided mask. An automatically cleared flag can't be cleared by this function.

Parameters

- base – FlexCAN peripheral base address.
- mask – The status flags to be cleared, it is logical OR value of _flexcan_flags.

static inline void FLEXCAN_GetBusErrCount(CAN_Type *base, uint8_t *txErrBuf, uint8_t *rxErrBuf)

Gets the FlexCAN Bus Error Counter value.

This function gets the FlexCAN Bus Error Counter value for both Tx and Rx direction. These values may be needed in the upper layer error handling.

Parameters

- base – FlexCAN peripheral base address.
- txErrBuf – Buffer to store Tx Error Counter value.
- rxErrBuf – Buffer to store Rx Error Counter value.

static inline uint64_t FLEXCAN_GetMbStatusFlags(CAN_Type *base, uint64_t mask)

Gets the FlexCAN low 64 Message Buffer interrupt flags.

This function gets the interrupt flags of a given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

Returns

The status of given Message Buffers.

static inline uint64_t FLEXCAN_GetHigh64MbStatusFlags(CAN_Type *base, uint64_t mask)

Gets the FlexCAN High 64 Message Buffer interrupt flags.

Valid only if the number of available MBs exceeds 64.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

Returns

The status of given Message Buffers.

static inline void FLEXCAN_ClearMbStatusFlags(CAN_Type *base, uint64_t mask)

Clears the FlexCAN low 64 Message Buffer interrupt flags.

This function clears the interrupt flags of a given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

static inline void FLEXCAN_ClearHigh64MbStatusFlags(CAN_Type *base, uint64_t mask)

Clears the FlexCAN High 64 Message Buffer interrupt flags.

Valid only if the number of available MBs exceeds 64.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

void FLEXCAN_GetMemoryErrorReportStatus(CAN_Type *base, *flexcan_memory_error_report_status_t* *errorStatus)

Gets the FlexCAN Memory Error Report registers status.

This function gets the FlexCAN Memory Error Report registers status.

Parameters

- base – FlexCAN peripheral base address.
- errorStatus – Pointer to FlexCAN Memory Error Report registers status structure.

static inline uint8_t FLEXCAN_GetPNMatchCount(CAN_Type *base)

Gets the FlexCAN Number of Matches when in Pretended Networking.

This function gets the number of times a given message has matched the predefined filtering criteria for ID and/or PL before a wakeup event.

Parameters

- base – FlexCAN peripheral base address.

Returns

The number of received wake up msessages.

static inline uint32_t FLEXCAN_GetEnhancedFifoDataCount(CAN_Type *base)

Gets the number of FlexCAN Enhanced Rx FIFO available frames.

This function gets the number of CAN messages stored in the Enhanced Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.

Returns

The number of available CAN messages stored in the Enhanced Rx FIFO.

static inline void FLEXCAN_EnableInterrupts(CAN_Type *base, uint64_t mask)

Enables FlexCAN interrupts according to the provided mask.

This function enables the FlexCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members, see _flexcan_interrupt_enable.

Parameters

- base – FlexCAN peripheral base address.

- mask – The interrupts to enable. Logical OR of _flexcan_interrupt_enable.

static inline void FLEXCAN_DisableInterrupts(CAN_Type *base, uint64_t mask)

Disables FlexCAN interrupts according to the provided mask.

This function disables the FlexCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members, see _flexcan_interrupt_enable.

Parameters

- base – FlexCAN peripheral base address.
- mask – The interrupts to disable. Logical OR of _flexcan_interrupt_enable.

static inline void FLEXCAN_EnableMbInterrupts(CAN_Type *base, uint64_t mask)

Enables FlexCAN low 64 Message Buffer interrupts.

This function enables the interrupts of given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

static inline void FLEXCAN_EnableHigh64MbInterrupts(CAN_Type *base, uint64_t mask)

Enables FlexCAN high 64 Message Buffer interrupts.

Valid only if the number of available MBs exceeds 64.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

static inline void FLEXCAN_DisableMbInterrupts(CAN_Type *base, uint64_t mask)

Disables FlexCAN low 64 Message Buffer interrupts.

This function disables the interrupts of given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

static inline void FLEXCAN_DisableHigh64MbInterrupts(CAN_Type *base, uint64_t mask)

Disables FlexCAN high 64 Message Buffer interrupts.

Valid only if the number of available MBs exceeds 64.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

void FLEXCAN_EnableRxFifoDMA(CAN_Type *base, bool enable)

Enables or disables the FlexCAN Rx FIFO DMA request.

This function enables or disables the DMA feature of FlexCAN build-in Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.
- enable – true to enable, false to disable.

static inline uintptr_t FLEXCAN_GetRxFifoHeadAddr(CAN_Type *base)

Gets the Rx FIFO Head address.

This function returns the FlexCAN Rx FIFO Head address, which is mainly used for the DMA/eDMA use case.

Parameters

- base – FlexCAN peripheral base address.

Returns

FlexCAN Rx FIFO Head address.

static inline *status_t* FLEXCAN_Enable(CAN_Type *base, bool enable)

Enables or disables the FlexCAN module operation.

This function enables or disables the FlexCAN module.

Parameters

- base – FlexCAN base pointer.
- enable – true to enable, false to disable.

Returns

kStatus_Success Enable FlexCAN module successful kStatus_Timeout Timeout when wait for Low-Power Mode Acknowledge

status_t FLEXCAN_WriteTxMb(CAN_Type *base, uint8_t mbIdx, const *flexcan_frame_t* *pTxFrame)

Writes a FlexCAN Message to the Transmit Message Buffer.

This function writes a CAN Message to the specified Transmit Message Buffer and changes the Message Buffer state to start CAN Message transmit. After that the function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The FlexCAN Message Buffer index.
- pTxFrame – Pointer to CAN message frame to be sent.

Return values

- kStatus_Success – - Write Tx Message Buffer Successfully.
- kStatus_Fail – - Tx Message Buffer is currently in use.

status_t FLEXCAN_ReadRxMb(CAN_Type *base, uint8_t mbIdx, *flexcan_frame_t* *pRxFrame)

Reads a FlexCAN Message from Receive Message Buffer.

This function reads a CAN message from a specified Receive Message Buffer. The function fills a receive CAN message frame structure with just received data and activates the Message Buffer again. The function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The FlexCAN Message Buffer index.
- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – - Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow – - Rx Message Buffer is already overflowed and has been read successfully.

- kStatus_Fail – - Rx Message Buffer is empty.

status_t FLEXCAN_WriteFDTxMb(CAN_Type *base, uint8_t mbIdx, const *flexcan_fd_frame_t* *pTxFrame)

Writes a FlexCAN FD Message to the Transmit Message Buffer.

This function writes a CAN FD Message to the specified Transmit Message Buffer and changes the Message Buffer state to start CAN FD Message transmit. After that the function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The FlexCAN FD Message Buffer index.
- pTxFrame – Pointer to CAN FD message frame to be sent.

Return values

- kStatus_Success – - Write Tx Message Buffer Successfully.
- kStatus_Fail – - Tx Message Buffer is currently in use.

status_t FLEXCAN_ReadFDRxMb(CAN_Type *base, uint8_t mbIdx, *flexcan_fd_frame_t* *pRxFrame)

Reads a FlexCAN FD Message from Receive Message Buffer.

This function reads a CAN FD message from a specified Receive Message Buffer. The function fills a receive CAN FD message frame structure with just received data and activates the Message Buffer again. The function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- mbIdx – The FlexCAN FD Message Buffer index.
- pRxFrame – Pointer to CAN FD message frame structure for reception.

Return values

- kStatus_Success – - Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow – - Rx Message Buffer is already overflowed and has been read successfully.
- kStatus_Fail – - Rx Message Buffer is empty.

status_t FLEXCAN_ReadRxFifo(CAN_Type *base, *flexcan_frame_t* *pRxFrame)

Reads a FlexCAN Message from Legacy Rx FIFO.

This function reads a CAN message from the FlexCAN Legacy Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.
- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – - Read Message from Rx FIFO successfully.
- kStatus_Fail – - Rx FIFO is not enabled.

status_t FLEXCAN_ReadEnhancedRxFifo(CAN_Type *base, *flexcan_fd_frame_t* *pRxFrame)

Reads a FlexCAN Message from Enhanced Rx FIFO.

This function reads a CAN or CAN FD message from the FlexCAN Enhanced Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.
- pRxFrame – Pointer to CAN FD message frame structure for reception.

Return values

- kStatus_Success – - Read Message from Rx FIFO successfully.
- kStatus_Fail – - Rx FIFO is not enabled.

status_t FLEXCAN_ReadPNWakeUpMB(CAN_Type *base, uint8_t mbIdx, *flexcan_frame_t* *pRxFrame)

Reads a FlexCAN Message from Wake Up MB.

This function reads a CAN message from the FlexCAN Wake up Message Buffers. There are four Wake up Message Buffers (WMBs) used to store incoming messages in Pretended Networking mode. The WMB index indicates the arrival order. The last message is stored in WMB3.

Parameters

- base – FlexCAN peripheral base address.
- pRxFrame – Pointer to CAN message frame structure for reception.
- mbIdx – The FlexCAN Wake up Message Buffer index. Range in 0x0 ~ 0x3.

Return values

- kStatus_Success – - Read Message from Wake up Message Buffer successfully.
- kStatus_Fail – - Wake up Message Buffer has no valid content.

status_t FLEXCAN_TransferFDSendBlocking(CAN_Type *base, uint8_t mbIdx, *flexcan_fd_frame_t* *pTxFrame)

Performs a polling send transaction on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- mbIdx – The FlexCAN FD Message Buffer index.
- pTxFrame – Pointer to CAN FD message frame to be sent.

Return values

- kStatus_Success – - Write Tx Message Buffer Successfully.
- kStatus_Fail – - Tx Message Buffer is currently in use.
- kStatus_Timeout – - Failed to send frames within specific time.

status_t FLEXCAN_TransferFDReceiveBlocking(CAN_Type *base, uint8_t mbIdx, *flexcan_fd_frame_t* *pRxFrame)

Performs a polling receive transaction on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.

- mbIdx – The FlexCAN FD Message Buffer index.
- pRxFrame – Pointer to CAN FD message frame structure for reception.

Return values

- kStatus_Success – - Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow – - Rx Message Buffer is already overflowed and has been read successfully.
- kStatus_Fail – - Rx Message Buffer is empty.
- kStatus_Timeout – - Failed to receive frames within specific time.

status_t FLEXCAN_TransferFDSendNonBlocking(CAN_Type *base, *flexcan_handle_t* *handle, *flexcan_mb_transfer_t* *pMbXfer)

Sends a message using IRQ.

This function sends a message using IRQ. This is a non-blocking function, which returns right away. When messages have been sent out, the send callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pMbXfer – FlexCAN FD Message Buffer transfer structure. See the flexcan_mb_transfer_t.

Return values

- kStatus_Success – Start Tx Message Buffer sending process successfully.
- kStatus_Fail – Write Tx Message Buffer failed.
- kStatus_FLEXCAN_TxBusy – Tx Message Buffer is in use.

status_t FLEXCAN_TransferFDReceiveNonBlocking(CAN_Type *base, *flexcan_handle_t* *handle, *flexcan_mb_transfer_t* *pMbXfer)

Receives a message using IRQ.

This function receives a message using IRQ. This is non-blocking function, which returns right away. When the message has been received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pMbXfer – FlexCAN FD Message Buffer transfer structure. See the flexcan_mb_transfer_t.

Return values

- kStatus_Success – - Start Rx Message Buffer receiving process successfully.
- kStatus_FLEXCAN_RxBusy – - Rx Message Buffer is in use.

void FLEXCAN_TransferFDAbortSend(CAN_Type *base, *flexcan_handle_t* *handle, uint8_t mbIdx)

Aborts the interrupt driven message send process.

This function aborts the interrupt driven message send process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

- mbIdx – The FlexCAN FD Message Buffer index.

void FLEXCAN_TransferFDAbortReceive(CAN_Type *base, *flexcan_handle_t* *handle, uint8_t mbIdx)

Aborts the interrupt driven message receive process.

This function aborts the interrupt driven message receive process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- mbIdx – The FlexCAN FD Message Buffer index.

status_t FLEXCAN_TransferSendBlocking(CAN_Type *base, uint8_t mbIdx, *flexcan_frame_t* *pTxFrame)

Performs a polling send transaction on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- mbIdx – The FlexCAN Message Buffer index.
- pTxFrame – Pointer to CAN message frame to be sent.

Return values

- kStatus_Success – - Write Tx Message Buffer Successfully.
- kStatus_Fail – - Tx Message Buffer is currently in use.
- kStatus_Timeout – - Failed to send frames within specific time.

status_t FLEXCAN_TransferReceiveBlocking(CAN_Type *base, uint8_t mbIdx, *flexcan_frame_t* *pRxFrame)

Performs a polling receive transaction on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- mbIdx – The FlexCAN Message Buffer index.
- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – - Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow – - Rx Message Buffer is already overflowed and has been read successfully.
- kStatus_Fail – - Rx Message Buffer is empty.
- kStatus_Timeout – - Failed to receive frames within specific time.

status_t FLEXCAN_TransferReceiveFifoBlocking(CAN_Type *base, *flexcan_frame_t* *pRxFrame)

Performs a polling receive transaction from Legacy Rx FIFO on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- pRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – - Read Message from Rx FIFO successfully.
- kStatus_Fail – - Rx FIFO is not enabled.
- kStatus_Timeout – - Failed to receive frames within specific time.

status_t FLEXCAN_TransferReceiveEnhancedFifoBlocking(CAN_Type *base, *flexcan_fd_frame_t* *pRxFrame)

Performs a polling receive transaction from Enhanced Rx FIFO on the CAN bus.

Note: A transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- pRxFrame – Pointer to CAN FD message frame structure for reception.

Return values

- kStatus_Success – - Read Message from Rx FIFO successfully.
- kStatus_Fail – - Rx FIFO is not enabled.
- kStatus_Timeout – - Failed to receive frames within specific time.

void FLEXCAN_TransferCreateHandle(CAN_Type *base, *flexcan_handle_t* *handle, *flexcan_transfer_callback_t* callback, void *userData)

Initializes the FlexCAN handle.

This function initializes the FlexCAN handle, which can be used for other FlexCAN transactional APIs. Usually, for a specified FlexCAN instance, call this API once to get the initialized handle.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- callback – The callback function.
- userData – The parameter of the callback function.

status_t FLEXCAN_TransferSendNonBlocking(CAN_Type *base, *flexcan_handle_t* *handle, *flexcan_mb_transfer_t* *pMbXfer)

Sends a message using IRQ.

This function sends a message using IRQ. This is a non-blocking function, which returns right away. When messages have been sent out, the send callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pMbXfer – FlexCAN Message Buffer transfer structure. See the flexcan_mb_transfer_t.

Return values

- kStatus_Success – Start Tx Message Buffer sending process successfully.
- kStatus_Fail – Write Tx Message Buffer failed.
- kStatus_FLEXCAN_TxBusy – Tx Message Buffer is in use.

status_t FLEXCAN_TransferReceiveNonBlocking(CAN_Type *base, *flexcan_handle_t* *handle, *flexcan_mb_transfer_t* *pMbXfer)

Receives a message using IRQ.

This function receives a message using IRQ. This is non-blocking function, which returns right away. When the message has been received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pMbXfer – FlexCAN Message Buffer transfer structure. See the flexcan_mb_transfer_t.

Return values

- kStatus_Success – - Start Rx Message Buffer receiving process successfully.
- kStatus_FLEXCAN_RxBusy – - Rx Message Buffer is in use.

status_t FLEXCAN_TransferReceiveFifoNonBlocking(CAN_Type *base, *flexcan_handle_t* *handle, *flexcan_fifo_transfer_t* *pFifoXfer)

Receives a message from Rx FIFO using IRQ.

This function receives a message using IRQ. This is a non-blocking function, which returns right away. When all messages have been received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pFifoXfer – FlexCAN Rx FIFO transfer structure. See the flexcan_fifo_transfer_t.

Return values

- kStatus_Success – - Start Rx FIFO receiving process successfully.
- kStatus_FLEXCAN_RxFifoBusy – - Rx FIFO is currently in use.

status_t FLEXCAN_TransferGetReceiveFifoCount(CAN_Type *base, *flexcan_handle_t* *handle, size_t *count)

Gets the Legacy Rx Fifo transfer status during a interrupt non-blocking receive.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- count – Number of CAN messages receive so far by the non-blocking transaction.

Return values

- kStatus_InvalidArgument – count is Invalid.
- kStatus_Success – Successfully return the count.

status_t FLEXCAN_TransferReceiveEnhancedFifoNonBlocking(CAN_Type *base, *flexcan_handle_t* *handle, *flexcan_fifo_transfer_t* *pFifoXfer)

Receives a message from Enhanced Rx FIFO using IRQ.

This function receives a message using IRQ. This is a non-blocking function, which returns right away. When all messages have been received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- pFifoXfer – FlexCAN Rx FIFO transfer structure. See the ref flexcan_fifo_transfer_t.@

Return values

- kStatus_Success – - Start Rx FIFO receiving process successfully.
- kStatus_FLEXCAN_RxFifoBusy – - Rx FIFO is currently in use.

static inline *status_t* FLEXCAN_TransferGetReceiveEnhancedFifoCount(CAN_Type *base, *flexcan_handle_t* *handle, size_t *count)

Gets the Enhanced Rx Fifo transfer status during a interrupt non-blocking receive.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- count – Number of CAN messages receive so far by the non-blocking transaction.

Return values

- kStatus_InvalidArgument – count is Invalid.
- kStatus_Success – Successfully return the count.

uint32_t FLEXCAN_GetTimeStamp(*flexcan_handle_t* *handle, uint8_t mbIdx)

Gets the detail index of Mailbox's Timestamp by handle.

Then function can only be used when calling non-blocking Data transfer (TX/RX) API, After TX/RX data transfer done (User can get the status by handler's callback function), we can get the detail index of Mailbox's timestamp by handle, Detail non-blocking data transfer API (TX/RX) contain. -FLEXCAN_TransferSendNonBlocking -FLEXCAN_TransferFDSendNonBlocking -FLEXCAN_TransferReceiveNonBlocking -FLEXCAN_TransferFDReceiveNonBlocking -FLEXCAN_TransferReceiveFifoNonBlocking

Parameters

- handle – FlexCAN handle pointer.
- mbIdx – The FlexCAN Message Buffer index.

Return values

the – index of mailbox 's timestamp stored in the handle.

static inline uint32_t FLEXCAN_GetHighResolutionTimeStamp(CAN_Type *base, uint8_t mbIdx)

void FLEXCAN_TransferAbortSend(CAN_Type *base, *flexcan_handle_t* *handle, uint8_t mbIdx)

Aborts the interrupt driven message send process.

This function aborts the interrupt driven message send process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- mbIdx – The FlexCAN Message Buffer index.

void FLEXCAN_TransferAbortReceive(CAN_Type *base, *flexcan_handle_t* *handle, uint8_t mbIdx)

Aborts the interrupt driven message receive process.

This function aborts the interrupt driven message receive process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- mbIdx – The FlexCAN Message Buffer index.

void FLEXCAN_TransferAbortReceiveFifo(CAN_Type *base, *flexcan_handle_t* *handle)

Aborts the interrupt driven message receive from Rx FIFO process.

This function aborts the interrupt driven message receive from Rx FIFO process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

void FLEXCAN_TransferAbortReceiveEnhancedFifo(CAN_Type *base, *flexcan_handle_t* *handle)

Aborts the interrupt driven message receive from Enhanced Rx FIFO process.

This function aborts the interrupt driven message receive from Enhanced Rx FIFO process.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

void FLEXCAN_TransferHandleIRQ(CAN_Type *base, *flexcan_handle_t* *handle)

FlexCAN IRQ handle function.

This function handles the FlexCAN Error, the Message Buffer, and the Rx FIFO IRQ request.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

void FLEXCAN_MbHandleIRQ(CAN_Type *base, *flexcan_handle_t* *handle, uint32_t startMbIdx, uint32_t endMbIdx)

FlexCAN Message Buffer IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- startMbIdx – First Message Buffer to handle.
- endMbIdx – Last Message Buffer to handle.

void FLEXCAN_EhancedRxFifoHandleIRQ(CAN_Type *base, *flexcan_handle_t* *handle)

FlexCAN Ehanced Rx FIFO IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

void FLEXCAN_BusoffErrorHandleIRQ(CAN_Type *base, *flexcan_handle_t* *handle)

FlexCAN Bus Off, Error and Warning IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

void FLEXCAN_PNWakeUpHandleIRQ(CAN_Type *base, *flexcan_handle_t* *handle)

FlexCAN Pretended Networking Wake-up IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

void FLEXCAN_MemoryErrorHandleIRQ(CAN_Type *base, *flexcan_handle_t* *handle)

FlexCAN Memory Error IRQ handle function.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.

FSL_FLEXCAN_DRIVER_VERSION

FlexCAN driver version.

FlexCAN transfer status.

Values:

enumerator kStatus_FLEXCAN_TxBusy

Tx Message Buffer is Busy.

enumerator kStatus_FLEXCAN_TxIdle

Tx Message Buffer is Idle.

enumerator kStatus_FLEXCAN_TxSwitchToRx

Remote Message is send out and Message buffer changed to Receive one.

enumerator kStatus_FLEXCAN_RxBusy

Rx Message Buffer is Busy.

enumerator kStatus_FLEXCAN_RxIdle

Rx Message Buffer is Idle.

enumerator kStatus_FLEXCAN_RxOverflow

Rx Message Buffer is Overflowed.

enumerator kStatus_FLEXCAN_RxFifoBusy

Rx Message FIFO is Busy.

enumerator kStatus_FLEXCAN_RxFifoIdle

Rx Message FIFO is Idle.

enumerator kStatus_FLEXCAN_RxFifoOverflow
Rx Message FIFO is overflowed.

enumerator kStatus_FLEXCAN_RxFifoWarning
Rx Message FIFO is almost overflowed.

enumerator kStatus_FLEXCAN_RxFifoDisabled
Rx Message FIFO is disabled during reading.

enumerator kStatus_FLEXCAN_ErrorStatus
FlexCAN Module Error and Status.

enumerator kStatus_FLEXCAN_WakeUp
FlexCAN is waken up from STOP mode.

enumerator kStatus_FLEXCAN_UnHandled
UnHadled Interrupt asserted.

enumerator kStatus_FLEXCAN_RxRemote
Rx Remote Message Received in Mail box.

enumerator kStatus_FLEXCAN_RxFifoUnderflow
Enhanced Rx Message FIFO is underflow.

enumerator kStatus_FLEXCAN_MemoryError
FlexCAN Memory Error.

enum _flexcan_frame_format
FlexCAN frame format.

Values:

enumerator kFLEXCAN_FrameFormatStandard
Standard frame format attribute.

enumerator kFLEXCAN_FrameFormatExtend
Extend frame format attribute.

enum _flexcan_frame_type
FlexCAN frame type.

Values:

enumerator kFLEXCAN_FrameTypeData
Data frame type attribute.

enumerator kFLEXCAN_FrameTypeRemote
Remote frame type attribute.

enum _flexcan_clock_source
FlexCAN clock source.

Deprecated:
Do not use the kFLEXCAN_ClkSrcOs. It has been superceded kFLEXCAN_ClkSrc0

Do not use the kFLEXCAN_ClkSrcPeri. It has been superceded kFLEXCAN_ClkSrc1

Values:

enumerator kFLEXCAN_ClkSrcOsc
FlexCAN Protocol Engine clock from Oscillator.

enumerator kFLEXCAN_ClkSrcPeri
FlexCAN Protocol Engine clock from Peripheral Clock.

enumerator kFLEXCAN_ClkSrc0
FlexCAN Protocol Engine clock selected by user as SRC == 0.

enumerator kFLEXCAN_ClkSrc1
FlexCAN Protocol Engine clock selected by user as SRC == 1.

enum _flexcan_wake_up_source
FlexCAN wake up source.

Values:

enumerator kFLEXCAN_WakeupSrcUnfiltered
FlexCAN uses unfiltered Rx input to detect edge.

enumerator kFLEXCAN_WakeupSrcFiltered
FlexCAN uses filtered Rx input to detect edge.

enum _flexcan_endianness
FlexCAN payload endianness.

Values:

enumerator kFLEXCAN_bigEndian
Transmit frame with MSB first, receive frame with big-endian format.

enumerator kFLEXCAN_littleEndian
Transmit frame with LSB first, receive frame with little-endian format.

enum _flexcan_MB_timestamp_base
FlexCAN timebase used for capturing 16-bit TIME_STAMP field of message buffer.

Values:

enumerator kFLEXCAN_CANTimer
FlexCAN free-running timer.

enumerator kFLEXCAN_Lower16bitsHRTimer
Lower 16 bits of high-resolution on-chip timer.

enumerator kFLEXCAN_Upper16bitsHRTimer
Upper 16 bits of high-resolution on-chip timer.

enum _flexcan_capture_point
FlexCAN capture point of 32-bit high resolution timebase during a CAN frame.

Values:

enumerator kFLEXCAN_CANFrameID2ndBit
Second bit of identifier field of any frame is on the CAN bus. HR_TIME_STAMPn register will not capture 32-bit counter value.

enumerator kFLEXCAN_CANFrameEnd
End of the CAN frame.

enumerator kFLEXCAN_CANFrameStart
Start of the CAN frame.

enumerator kFLEXCAN_CANFDFrameRes
Start of frame for classical CAN frames; res bit for CAN FD frames.

enum _flexcan_rx_fifo_filter_type

FlexCAN Rx Fifo Filter type.

Values:

enumerator kFLEXCAN_RxFifoFilterTypeA

One full ID (standard and extended) per ID Filter element.

enumerator kFLEXCAN_RxFifoFilterTypeB

Two full standard IDs or two partial 14-bit ID slices per ID Filter Table element.

enumerator kFLEXCAN_RxFifoFilterTypeC

Four partial 8-bit Standard or extended ID slices per ID Filter Table element.

enumerator kFLEXCAN_RxFifoFilterTypeD

All frames rejected.

enum _flexcan_mb_size

FlexCAN Message Buffer Payload size.

Values:

enumerator kFLEXCAN_8BperMB

Selects 8 bytes per Message Buffer.

enumerator kFLEXCAN_16BperMB

Selects 16 bytes per Message Buffer.

enumerator kFLEXCAN_32BperMB

Selects 32 bytes per Message Buffer.

enumerator kFLEXCAN_64BperMB

Selects 64 bytes per Message Buffer.

enum _flexcan_fd_frame_length

FlexCAN CAN FD frame supporting data length (available DLC values).

For Tx, when the Data size corresponding to DLC value stored in the MB selected for transmission is larger than the MB Payload size, FlexCAN adds the necessary number of bytes with constant 0xCC pattern to complete the expected DLC. For Rx, when the Data size corresponding to DLC value received from the CAN bus is larger than the MB Payload size, the high order bytes that do not fit the Payload size will lose.

Values:

enumerator kFLEXCAN_0BperFrame

Frame contains 0 valid data bytes.

enumerator kFLEXCAN_1BperFrame

Frame contains 1 valid data bytes.

enumerator kFLEXCAN_2BperFrame

Frame contains 2 valid data bytes.

enumerator kFLEXCAN_3BperFrame

Frame contains 3 valid data bytes.

enumerator kFLEXCAN_4BperFrame

Frame contains 4 valid data bytes.

enumerator kFLEXCAN_5BperFrame

Frame contains 5 valid data bytes.

enumerator kFLEXCAN_6BperFrame
Frame contains 6 valid data bytes.

enumerator kFLEXCAN_7BperFrame
Frame contains 7 valid data bytes.

enumerator kFLEXCAN_8BperFrame
Frame contains 8 valid data bytes.

enumerator kFLEXCAN_12BperFrame
Frame contains 12 valid data bytes.

enumerator kFLEXCAN_16BperFrame
Frame contains 16 valid data bytes.

enumerator kFLEXCAN_20BperFrame
Frame contains 20 valid data bytes.

enumerator kFLEXCAN_24BperFrame
Frame contains 24 valid data bytes.

enumerator kFLEXCAN_32BperFrame
Frame contains 32 valid data bytes.

enumerator kFLEXCAN_48BperFrame
Frame contains 48 valid data bytes.

enumerator kFLEXCAN_64BperFrame
Frame contains 64 valid data bytes.

enum _flexcan_efifo_dma_per_read_length
FlexCAN Enhanced Rx Fifo DMA transfer per read length enumerations.

Values:

enumerator kFLEXCAN_1WordPerRead
Transfer 1 32-bit words (CS).

enumerator kFLEXCAN_2WordPerRead
Transfer 2 32-bit words (CS + ID).

enumerator kFLEXCAN_3WordPerRead
Transfer 3 32-bit words (CS + ID + 1~4 bytes data).

enumerator kFLEXCAN_4WordPerRead
Transfer 4 32-bit words (CS + ID + 5~8 bytes data).

enumerator kFLEXCAN_5WordPerRead
Transfer 5 32-bit words (CS + ID + 9~12 bytes data).

enumerator kFLEXCAN_6WordPerRead
Transfer 6 32-bit words (CS + ID + 13~16 bytes data).

enumerator kFLEXCAN_7WordPerRead
Transfer 7 32-bit words (CS + ID + 17~20 bytes data).

enumerator kFLEXCAN_8WordPerRead
Transfer 8 32-bit words (CS + ID + 21~24 bytes data).

enumerator kFLEXCAN_9WordPerRead
Transfer 9 32-bit words (CS + ID + 25~28 bytes data).

enumerator kFLEXCAN_10WordPerRead
Transfer 10 32-bit words (CS + ID + 29~32 bytes data).

enumerator kFLEXCAN_11WordPerRead
Transfer 11 32-bit words (CS + ID + 33~36 bytes data).

enumerator kFLEXCAN_12WordPerRead
Transfer 12 32-bit words (CS + ID + 37~40 bytes data).

enumerator kFLEXCAN_13WordPerRead
Transfer 13 32-bit words (CS + ID + 41~44 bytes data).

enumerator kFLEXCAN_14WordPerRead
Transfer 14 32-bit words (CS + ID + 45~48 bytes data).

enumerator kFLEXCAN_15WordPerRead
Transfer 15 32-bit words (CS + ID + 49~52 bytes data).

enumerator kFLEXCAN_16WordPerRead
Transfer 16 32-bit words (CS + ID + 53~56 bytes data).

enumerator kFLEXCAN_17WordPerRead
Transfer 17 32-bit words (CS + ID + 57~60 bytes data).

enumerator kFLEXCAN_18WordPerRead
Transfer 18 32-bit words (CS + ID + 61~64 bytes data).

enumerator kFLEXCAN_19WordPerRead
Transfer 19 32-bit words (CS + ID + 64 bytes data + ID HIT).

enumerator kFLEXCAN_20WordPerRead
Transfer 20 32-bit words (CS + ID + 64 bytes data + ID HIT + HR timestamp).

enum _flexcan_rx_fifo_priority
FlexCAN Enhanced/Legacy Rx FIFO priority.

The matching process starts from the Rx MB(or Enhanced/Legacy Rx FIFO) with higher priority. If no MB(or Enhanced/Legacy Rx FIFO filter) is satisfied, the matching process goes on with the Enhanced/Legacy Rx FIFO(or Rx MB) with lower priority.

Values:

enumerator kFLEXCAN_RxFifoPrioLow
Matching process start from Rx Message Buffer first.

enumerator kFLEXCAN_RxFifoPrioHigh
Matching process start from Enhanced/Legacy Rx FIFO first.

enum _flexcan_interrupt_enable
FlexCAN interrupt enable enumerations.

This provides constants for the FlexCAN interrupt enable enumerations for use in the FlexCAN functions.

Note: FlexCAN Message Buffers and Legacy Rx FIFO interrupts not included in.

Values:

enumerator kFLEXCAN_BusOffInterruptEnable
Bus Off interrupt, use bit 15.

enumerator kFLEXCAN_ErrorInterruptEnable
CAN Error interrupt, use bit 14.

enumerator kFLEXCAN_TxWarningInterruptEnable
Tx Warning interrupt, use bit 11.

enumerator kFLEXCAN_RxWarningInterruptEnable
Rx Warning interrupt, use bit 10.

enumerator kFLEXCAN_FDErrorInterruptEnable
CAN FD Error interrupt, use bit 31.

enumerator kFLEXCAN_PNMatchWakeUpInterruptEnable
PN Match Wake Up interrupt, use high word bit 17.

enumerator kFLEXCAN_PNTimeoutWakeUpInterruptEnable
PN Timeout Wake Up interrupt, use high word bit 16. Enhanced Rx FIFO Underflow interrupt, use high word bit 31.

enumerator kFLEXCAN_ERxFifoUnderflowInterruptEnable
Enhanced Rx FIFO Overflow interrupt, use high word bit 30.

enumerator kFLEXCAN_ERxFifoOverflowInterruptEnable
Enhanced Rx FIFO Watermark interrupt, use high word bit 29.

enumerator kFLEXCAN_ERxFifoWatermarkInterruptEnable
Enhanced Rx FIFO Data Avilable interrupt, use high word bit 28.

enumerator kFLEXCAN_ERxFifoDataAvlInterruptEnable

enumerator kFLEXCAN_HostAccessNCErrorInterruptEnable
Host Access With Non-Correctable Errors interrupt, use high word bit 0.

enumerator kFLEXCAN_FlexCanAccessNCErrorInterruptEnable
FlexCAN Access With Non-Correctable Errors interrupt, use high word bit 2.

enumerator kFLEXCAN_HostOrFlexCanCErrorInterruptEnable
Host or FlexCAN Access With Correctable Errors interrupt, use high word bit 3.

enum _flexcan_flags
FlexCAN status flags.

This provides constants for the FlexCAN status flags for use in the FlexCAN functions.

Note: The CPU read action clears the bits corresponding to the FlEXCAN_ErrorFlag macro, therefore user need to read status flags and distinguish which error is occur using _flexcan_error_flags enumerations.

Values:

enumerator kFLEXCAN_ErrorOverrunFlag
Error Overrun Status.

enumerator kFLEXCAN_FDErrorIntFlag
CAN FD Error Interrupt Flag.

enumerator kFLEXCAN_BusoffDoneIntFlag
Bus Off process completed Interrupt Flag.

enumerator kFLEXCAN_SynchFlag
CAN Synchronization Status.

enumerator kFLEXCAN_TxWarningIntFlag
Tx Warning Interrupt Flag.

enumerator kFLEXCAN_RxWarningIntFlag
Rx Warning Interrupt Flag.

enumerator kFLEXCAN_IdleFlag
FlexCAN In IDLE Status.

enumerator kFLEXCAN_FaultConfinementFlag
FlexCAN Fault Confinement State.

enumerator kFLEXCAN_TransmittingFlag
FlexCAN In Transmission Status.

enumerator kFLEXCAN_ReceivingFlag
FlexCAN In Reception Status.

enumerator kFLEXCAN_BusOffIntFlag
Bus Off Interrupt Flag.

enumerator kFLEXCAN_ErrorIntFlag
CAN Error Interrupt Flag.

enumerator kFLEXCAN_ErrorFlag

enumerator kFLEXCAN_PNMatchIntFlag
PN Matching Event Interrupt Flag.

enumerator kFLEXCAN_PNTimeoutIntFlag
PN Timeout Event Interrupt Flag.

enumerator kFLEXCAN_ERxFifoUnderflowIntFlag
Enhanced Rx FIFO underflow Interrupt Flag.

enumerator kFLEXCAN_ERxFifoOverflowIntFlag
Enhanced Rx FIFO overflow Interrupt Flag.

enumerator kFLEXCAN_ERxFifoWatermarkIntFlag
Enhanced Rx FIFO watermark Interrupt Flag.

enumerator kFLEXCAN_ERxFifoDataAvlIntFlag
Enhanced Rx FIFO data available Interrupt Flag.

enumerator kFLEXCAN_ERxFifoEmptyFlag
Enhanced Rx FIFO empty status.

enumerator kFLEXCAN_ERxFifoFullFlag
Enhanced Rx FIFO full status.

enumerator kFLEXCAN_HostAccessNonCorrectableErrorIntFlag
Host Access With Non-Correctable Error Interrupt Flag.

enumerator kFLEXCAN_FlexCanAccessNonCorrectableErrorIntFlag
FlexCAN Access With Non-Correctable Error Interrupt Flag.

enumerator kFLEXCAN_CorrectableErrorIntFlag
Correctable Error Interrupt Flag.

enumerator kFLEXCAN_HostAccessNonCorrectableErrorOverrunFlag
Host Access With Non-Correctable Error Interrupt Overrun Flag.

enumerator kFLEXCAN_FlexCanAccessNonCorrectableErrorOverrunFlag
FlexCAN Access With Non-Correctable Error Interrupt Overrun Flag.

enumerator kFLEXCAN_CorrectableErrorOverrunFlag
Correctable Error Interrupt Overrun Flag.

enumerator kFLEXCAN_AllMemoryErrorIntFlag
All Memory Error Interrupt Flags.

enumerator kFLEXCAN_AllMemoryErrorFlag
All Memory Error Flags.

enum _flexcan_error_flags
FlexCAN error status flags.

The FlexCAN Error Status enumerations is used to report current error of the FlexCAN bus. This enumerations should be used with KFLEXCAN_ErrorFlag in _flexcan_flags enumerations to ditermine which error is generated.

Values:

enumerator kFLEXCAN_FDStuffingError
Stuffing Error.

enumerator kFLEXCAN_FDFormError
Form Error.

enumerator kFLEXCAN_FDCrcError
Cyclic Redundancy Check Error.

enumerator kFLEXCAN_FDBit0Error
Unable to send dominant bit.

enumerator kFLEXCAN_FDBit1Error
Unable to send recessive bit.

enumerator kFLEXCAN_TxErrorWarningFlag
Tx Error Warning Status.

enumerator kFLEXCAN_RxErrorWarningFlag
Rx Error Warning Status.

enumerator kFLEXCAN_StuffingError
Stuffing Error.

enumerator kFLEXCAN_FormError
Form Error.

enumerator kFLEXCAN_CrcError
Cyclic Redundancy Check Error.

enumerator kFLEXCAN_AckError
Received no ACK on transmission.

enumerator kFLEXCAN_Bit0Error
Unable to send dominant bit.

enumerator kFLEXCAN_Bit1Error
Unable to send recessive bit.

FlexCAN Legacy Rx FIFO status flags.

The FlexCAN Legacy Rx FIFO Status enumerations are used to determine the status of the Rx FIFO. Because Rx FIFO occupy the MB0 ~ MB7 (Rx Fifo filter also occupies more Message Buffer space), Rx FIFO status flags are mapped to the corresponding Message Buffer status flags.

Values:

enumerator kFLEXCAN_RxFifoOverflowFlag
Rx FIFO overflow flag.

enumerator kFLEXCAN_RxFifoWarningFlag
Rx FIFO almost full flag.

enumerator kFLEXCAN_RxFifoFrameAvlFlag
Frames available in Rx FIFO flag.

enum _flexcan_memory_error_type
FlexCAN Memory Error Type.

Values:

enumerator kFLEXCAN_CorrectableError
The memory error is correctable which means on bit error.

enumerator kFLEXCAN_NonCorrectableError
The memory error is non-correctable which means two bit errors.

enum _flexcan_memory_access_type
FlexCAN Memory Access Type.

Values:

enumerator kFLEXCAN_MoveOutFlexCanAccess
The memory error was detected during move-out FlexCAN access.

enumerator kFLEXCAN_MoveInAccess
The memory error was detected during move-in FlexCAN access.

enumerator kFLEXCAN_TxArbitrationAccess
The memory error was detected during Tx Arbitration FlexCAN access.

enumerator kFLEXCAN_RxMatchingAccess
The memory error was detected during Rx Matching FlexCAN access.

enumerator kFLEXCAN_MoveOutHostAccess
The memory error was detected during Rx Matching Host (CPU) access.

enum _flexcan_byte_error_syndrome
FlexCAN Memory Error Byte Syndrome.

Values:

enumerator kFLEXCAN_NoError
No bit error in this byte.

enumerator kFLEXCAN_ParityBits0Error
Parity bit 0 error in this byte.

enumerator kFLEXCAN_ParityBits1Error
Parity bit 1 error in this byte.

enumerator kFLEXCAN_ParityBits2Error
Parity bit 2 error in this byte.

enumerator kFLEXCAN_ParityBits3Error
Parity bit 3 error in this byte.

enumerator kFLEXCAN_ParityBits4Error
Parity bit 4 error in this byte.

enumerator kFLEXCAN_DataBits0Error
Data bit 0 error in this byte.

enumerator kFLEXCAN_DataBits1Error
Data bit 1 error in this byte.

enumerator kFLEXCAN_DataBits2Error
Data bit 2 error in this byte.

enumerator kFLEXCAN_DataBits3Error
Data bit 3 error in this byte.

enumerator kFLEXCAN_DataBits4Error
Data bit 4 error in this byte.

enumerator kFLEXCAN_DataBits5Error
Data bit 5 error in this byte.

enumerator kFLEXCAN_DataBits6Error
Data bit 6 error in this byte.

enumerator kFLEXCAN_DataBits7Error
Data bit 7 error in this byte.

enumerator kFLEXCAN_AllZeroError
All-zeros non-correctable error in this byte.

enumerator kFLEXCAN_AllOneError
All-ones non-correctable error in this byte.

enumerator kFLEXCAN_NonCorrectableErrors
Non-correctable error in this byte.

enum _flexcan_pn_match_source
FlexCAN Pretended Networking match source selection.

Values:

enumerator kFLEXCAN_PNMatSrcID
Message match with ID filtering.

enumerator kFLEXCAN_PNMatSrcIDAndData
Message match with ID filtering and payload filtering.

enum _flexcan_pn_match_mode
FlexCAN Pretended Networking mode match type.

Values:

enumerator kFLEXCAN_PNMatModeEqual
Match upon ID/Payload contents against an exact target value.

enumerator kFLEXCAN_PNMatModeGreater
Match upon an ID/Payload value greater than or equal to a specified target value.

enumerator kFLEXCAN_PNMatModeSmaller

Match upon an ID/Payload value smaller than or equal to a specified target value.

enumerator kFLEXCAN_PNMatModeRange

Match upon an ID/Payload value inside a range, greater than or equal to a specified lower limit, and smaller than or equal to a specified upper limit

typedef enum *_flexcan_frame_format* flexcan_frame_format_t

FlexCAN frame format.

typedef enum *_flexcan_frame_type* flexcan_frame_type_t

FlexCAN frame type.

typedef enum *_flexcan_clock_source* flexcan_clock_source_t

FlexCAN clock source.

Deprecated:

Do not use the kFLEXCAN_ClkSrcOs. It has been superceded kFLEXCAN_ClkSrc0

Do not use the kFLEXCAN_ClkSrcPeri. It has been superceded kFLEXCAN_ClkSrc1

typedef enum *_flexcan_wake_up_source* flexcan_wake_up_source_t

FlexCAN wake up source.

typedef enum *_flexcan_endianness* flexcan_endianness_t

FlexCAN payload endianness.

typedef enum *_flexcan_MB_timestamp_base* flexcan_MB_timestamp_base_t

FlexCAN timebase used for capturing 16-bit TIME_STAMP field of message buffer.

typedef enum *_flexcan_capture_point* flexcan_capture_point_t

FlexCAN capture point of 32-bit high resolution timebase during a CAN frame.

typedef enum *_flexcan_rx_fifo_filter_type* flexcan_rx_fifo_filter_type_t

FlexCAN Rx Fifo Filter type.

typedef enum *_flexcan_mb_size* flexcan_mb_size_t

FlexCAN Message Buffer Payload size.

typedef enum *_flexcan_efifo_dma_per_read_length* flexcan_efifo_dma_per_read_length_t

FlexCAN Enhanced Rx Fifo DMA transfer per read length enumerations.

typedef enum *_flexcan_rx_fifo_priority* flexcan_rx_fifo_priority_t

FlexCAN Enhanced/Legacy Rx FIFO priority.

The matching process starts from the Rx MB(or Enhanced/Legacy Rx FIFO) with higher priority. If no MB(or Enhanced/Legacy Rx FIFO filter) is satisfied, the matching process goes on with the Enhanced/Legacy Rx FIFO(or Rx MB) with lower priority.

typedef enum *_flexcan_memory_error_type* flexcan_memory_error_type_t

FlexCAN Memory Error Type.

typedef enum *_flexcan_memory_access_type* flexcan_memory_access_type_t

FlexCAN Memory Access Type.

typedef enum *_flexcan_byte_error_syndrome* flexcan_byte_error_syndrome_t

FlexCAN Memory Error Byte Syndrome.

typedef struct *_flexcan_memory_error_report_status* flexcan_memory_error_report_status_t

FlexCAN memory error register status structure.

This structure contains the memory access properties that caused a memory error access. It is used as the parameter of FLEXCAN_GetMemoryErrorReportStatus() function. And user can use FLEXCAN_GetMemoryErrorReportStatus to get the status of the last memory error access.

typedef struct *_flexcan_frame* flexcan_frame_t

FlexCAN message frame structure.

typedef struct *_flexcan_fd_frame* flexcan_fd_frame_t

CAN FD message frame structure.

The CAN FD message supporting up to sixty four bytes can be used for a data frame, depending on the length selected for the message buffers. The length should be a enumeration member, see _flexcan_fd_frame_length.

typedef struct *_flexcan_timing_config* flexcan_timing_config_t

FlexCAN protocol timing characteristic configuration structure.

typedef struct *_flexcan_config* flexcan_config_t

FlexCAN module configuration structure.

Deprecated:

Do not use the baudRate. It has been superceded bitRate

Do not use the baudRateFD. It has been superceded bitRateFD

typedef struct *_flexcan_rx_mb_config* flexcan_rx_mb_config_t

FlexCAN Receive Message Buffer configuration structure.

This structure is used as the parameter of FLEXCAN_SetRxMbConfig() function. The FLEXCAN_SetRxMbConfig() function is used to configure FlexCAN Receive Message Buffer. The function abort previous receiving process, clean the Message Buffer and activate the Rx Message Buffer using given Message Buffer setting.

typedef enum *_flexcan_pn_match_source* flexcan_pn_match_source_t

FlexCAN Pretended Networking match source selection.

typedef enum *_flexcan_pn_match_mode* flexcan_pn_match_mode_t

FlexCAN Pretended Networking mode match type.

typedef struct *_flexcan_pn_config* flexcan_pn_config_t

FlexCAN Pretended Networking configuration structure.

This structure is used as the parameter of FLEXCAN_SetPNConfig() function. The FLEXCAN_SetPNConfig() function is used to configure FlexCAN Networking work mode.

typedef struct *_flexcan_rx_fifo_config* flexcan_rx_fifo_config_t

FlexCAN Legacy Rx FIFO configuration structure.

typedef struct *_flexcan_enhanced_rx_fifo_std_id_filter* flexcan_enhanced_rx_fifo_std_id_filter_t

FlexCAN Enhanced Rx FIFO Standard ID filter element structure.

typedef struct *_flexcan_enhanced_rx_fifo_ext_id_filter* flexcan_enhanced_rx_fifo_ext_id_filter_t

FlexCAN Enhanced Rx FIFO Extended ID filter element structure.

typedef struct *_flexcan_enhanced_rx_fifo_config* flexcan_enhanced_rx_fifo_config_t

FlexCAN Enhanced Rx FIFO configuration structure.

typedef struct *_flexcan_mb_transfer* flexcan_mb_transfer_t
FlexCAN Message Buffer transfer.

typedef struct *_flexcan_fifo_transfer* flexcan_fifo_transfer_t
FlexCAN Rx FIFO transfer.

typedef struct *_flexcan_handle* flexcan_handle_t
FlexCAN handle structure definition.

typedef void (*flexcan_transfer_callback_t)(CAN_Type *base, *flexcan_handle_t* *handle, *status_t* status, uint64_t result, void *userData)

FLEXCAN_WAIT_TIMEOUT

FLEXCAN_POLLING_TIMEOUT
Max loops to wait for polling transfer.

FLEXCAN_MODULE_TIMEOUT
Max loops to wait for FlexCAN register access complete.

DLC_LENGTH_DECODE(dlc)
FlexCAN frame length helper macro.

FLEXCAN_ID_STD(id)
FlexCAN Frame ID helper macro.

Standard Frame ID helper macro.

FLEXCAN_ID_EXT(id)
Extend Frame ID helper macro.

FLEXCAN_RX_MB_STD_MASK(id, rtr, ide)
FlexCAN Rx Message Buffer Mask helper macro.

Standard Rx Message Buffer Mask helper macro.

FLEXCAN_RX_MB_EXT_MASK(id, rtr, ide)
Extend Rx Message Buffer Mask helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_A(id, rtr, ide)
FlexCAN Legacy Rx FIFO Mask helper macro.

Standard Rx FIFO Mask helper macro Type A helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH(id, rtr, ide)
Standard Rx FIFO Mask helper macro Type B upper part helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW(id, rtr, ide)
Standard Rx FIFO Mask helper macro Type B lower part helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH(id)
Standard Rx FIFO Mask helper macro Type C upper part helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH(id)
Standard Rx FIFO Mask helper macro Type C mid-upper part helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW(id)
Standard Rx FIFO Mask helper macro Type C mid-lower part helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW(id)
Standard Rx FIFO Mask helper macro Type C lower part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(id, rtr, ide)
Extend Rx FIFO Mask helper macro Type A helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH(id, rtr, ide)
Extend Rx FIFO Mask helper macro Type B upper part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW(id, rtr, ide)
Extend Rx FIFO Mask helper macro Type B lower part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(id)
Extend Rx FIFO Mask helper macro Type C upper part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH(id)
Extend Rx FIFO Mask helper macro Type C mid-upper part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW(id)
Extend Rx FIFO Mask helper macro Type C mid-lower part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(id)
Extend Rx FIFO Mask helper macro Type C lower part helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(id, rtr, ide)
FlexCAN Rx FIFO Filter helper macro.

Standard Rx FIFO Filter helper macro Type A helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_HIGH(id, rtr, ide)
Standard Rx FIFO Filter helper macro Type B upper part helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_LOW(id, rtr, ide)
Standard Rx FIFO Filter helper macro Type B lower part helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_HIGH(id)
Standard Rx FIFO Filter helper macro Type C upper part helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_HIGH(id)
Standard Rx FIFO Filter helper macro Type C mid-upper part helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_LOW(id)
Standard Rx FIFO Filter helper macro Type C mid-lower part helper macro.

FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_LOW(id)
Standard Rx FIFO Filter helper macro Type C lower part helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_A(id, rtr, ide)
Extend Rx FIFO Filter helper macro Type A helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_HIGH(id, rtr, ide)
Extend Rx FIFO Filter helper macro Type B upper part helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_LOW(id, rtr, ide)
Extend Rx FIFO Filter helper macro Type B lower part helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_HIGH(id)
Extend Rx FIFO Filter helper macro Type C upper part helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_HIGH(id)
Extend Rx FIFO Filter helper macro Type C mid-upper part helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_LOW(id)
Extend Rx FIFO Filter helper macro Type C mid-lower part helper macro.

FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_LOW(id)
Extend Rx FIFO Filter helper macro Type C lower part helper macro.

ENHANCED_RX_FIFO_FSCH(x)
FlexCAN Enhanced Rx FIFO Filter and Mask helper macro.

RTR_STD_HIGH(x)

RTR_STD_LOW(x)

RTR_EXT(x)

ID_STD_LOW(id)

ID_STD_HIGH(id)

ID_EXT(id)

FLEXCAN_ENHANCED_RX_FIFO_STD_MASK_AND_FILTER(id, rtr, id_mask, rtr_mask)
Standard ID filter element with filter + mask scheme.

FLEXCAN_ENHANCED_RX_FIFO_STD_FILTER_WITH_RANGE(id_upper, rtr, id_lower, rtr_mask)
Standard ID filter element with filter range.

FLEXCAN_ENHANCED_RX_FIFO_STD_TWO_FILTERS(id1, rtr1, id2, rtr2)
Standard ID filter element with two filters without masks.

FLEXCAN_ENHANCED_RX_FIFO_EXT_MASK_AND_FILTER_LOW(id, rtr)
Extended ID filter element with filter + mask scheme low word.

FLEXCAN_ENHANCED_RX_FIFO_EXT_MASK_AND_FILTER_HIGH(id_mask, rtr_mask)
Extended ID filter element with filter + mask scheme high word.

FLEXCAN_ENHANCED_RX_FIFO_EXT_FILTER_WITH_RANGE_LOW(id_upper, rtr)
Extended ID filter element with range scheme low word.

FLEXCAN_ENHANCED_RX_FIFO_EXT_FILTER_WITH_RANGE_HIGH(id_lower, rtr_mask)
Extended ID filter element with range scheme high word.

FLEXCAN_ENHANCED_RX_FIFO_EXT_TWO_FILTERS_LOW(id2, rtr2)
Extended ID filter element with two filters without masks low word.

FLEXCAN_ENHANCED_RX_FIFO_EXT_TWO_FILTERS_HIGH(id1, rtr1)
Extended ID filter element with two filters without masks high word.

FLEXCAN_PN_STD_MASK(id, rtr)
FlexCAN Pretended Networking ID Mask helper macro.

Standard Rx Message Buffer Mask helper macro.

FLEXCAN_PN_EXT_MASK(id, rtr)
Extend Rx Message Buffer Mask helper macro.

FLEXCAN_PN_INT_MASK(x)
FlexCAN interrupt/status flag helper macro.

FLEXCAN_PN_INT_UNMASK(x)

FLEXCAN_PN_STATUS_MASK(x)

FLEXCAN_PN_STATUS_UNMASK(x)

FLEXCAN_EFIFO_INT_MASK(x)

FLEXCAN_EFIFO_INT_UNMASK(x)

FLEXCAN_EFIFO_STATUS_MASK(x)

FLEXCAN_EFIFO_STATUS_UNMASK(x)

FLEXCAN_MECR_INT_MASK(x)

FLEXCAN_MECR_INT_UNMASK(x)

FLEXCAN_MECR_STATUS_MASK(x)

FLEXCAN_MECR_STATUS_UNMASK(x)

FLEXCAN_ERROR_AND_STATUS_INT_FLAG

FLEXCAN_PNWAKE_UP_FLAG

FLEXCAN_WAKE_UP_FLAG

FLEXCAN_MEMORY_ERROR_INT_FLAG

FLEXCAN_ENHANCED_RX_FIFO_INT_FLAG

FlexCAN Enhanced Rx FIFO base address helper macro.

E_RX_FIFO(base)

FLEXCAN_CALLBACK(x)

FlexCAN transfer callback function.

The FlexCAN transfer callback returns a value from the underlying layer. If the status equals to kStatus_FLEXCAN_ErrorStatus, the result parameter is the Content of FlexCAN status register which can be used to get the working status(or error status) of FlexCAN module. If the status equals to other FlexCAN Message Buffer transfer status, the result is the index of Message Buffer that generate transfer event. If the status equals to other FlexCAN Message Buffer transfer status, the result is meaningless and should be Ignored.

struct __flexcan_memory_error_report_status

#include <fsl_flexcan.h> FlexCAN memory error register status structure.

This structure contains the memory access properties that caused a memory error access. It is used as the parameter of FLEXCAN_GetMemoryErrorReportStatus() function. And user can use FLEXCAN_GetMemoryErrorReportStatus to get the status of the last memory error access.

Public Members

flexcan_memory_error_type_t errorType

The type of memory error that giving rise to the report.

flexcan_memory_access_type_t accessType

The type of memory access that giving rise to the memory error.

uint16_t accessAddress

The address where memory error detected.

uint32_t errorData

The raw data word read from memory with error.

struct __flexcan_frame

#include <fsl_flexcan.h> FlexCAN message frame structure.

struct _flexcan_fd_frame

#include <fsl_flexcan.h> CAN FD message frame structure.

The CAN FD message supporting up to sixty four bytes can be used for a data frame, depending on the length selected for the message buffers. The length should be a enumeration member, see _flexcan_fd_frame_length.

Public Members

uint32_t idhit

Note: ID HIT offset is changed dynamically according to data length code (DLC), when DLC is 15, they will be located below. Using FLEXCAN_FixEnhancedRxFifoFrameIdHit API is recommended to ensure this idhit value is correct. CAN Enhanced Rx FIFO filter hit id (This value is only used in Enhanced Rx FIFO receive mode).

uint32_t hrtimestamp

Note: HR timestamp offset is changed dynamically according to data length code (DLC). External 32-bit on-chip timer high-resolution timestamp.

struct _flexcan_timing_config

#include <fsl_flexcan.h> FlexCAN protocol timing characteristic configuration structure.

Public Members

uint32_t preDivider

Classic CAN or CAN FD nominal phase bit rate prescaler.

uint32_t rJumpwidth

Classic CAN or CAN FD nominal phase Re-sync Jump Width.

uint32_t phaseSeg1

Classic CAN or CAN FD nominal phase Segment 1.

uint32_t phaseSeg2

Classic CAN or CAN FD nominal phase Segment 2.

uint32_t propSeg

Classic CAN or CAN FD nominal phase Propagation Segment.

uint32_t fpreDivider

CAN FD data phase bit rate prescaler.

uint32_t frJumpwidth

CAN FD data phase Re-sync Jump Width.

uint32_t fphaseSeg1

CAN FD data phase Phase Segment 1.

uint32_t fphaseSeg2

CAN FD data phase Phase Segment 2.

uint32_t fpropSeg

CAN FD data phase Propagation Segment.

struct _flexcan_config

#include <fsl_flexcan.h> FlexCAN module configuration structure.

Deprecated:

Do not use the baudRate. It has been superceded bitRate

Do not use the baudRateFD. It has been superceded bitRateFD

Public Members

flexcan_clock_source_t clkSrc

Clock source for FlexCAN Protocol Engine.

flexcan_wake_up_source_t wakeupSrc

Wake up source selection.

uint8_t maxMbNum

The maximum number of Message Buffers used by user.

bool enableLoopBack

Enable or Disable Loop Back Self Test Mode.

bool enableTimerSync

Enable or Disable Timer Synchronization.

bool enableIndividMask

Enable or Disable Rx Individual Mask and Queue feature.

bool disableSelfReception

Enable or Disable Self Reflection.

bool enableListenOnlyMode

Enable or Disable Listen Only Mode.

bool enableDoze

Enable or Disable Doze Mode.

bool enablePretendedeNetworking

Enable or Disable the Pretended Networking mode.

bool enableMemoryErrorControl

Enable or Disable the memory errors detection and correction mechanism.

bool enableNonCorrectableErrorEnterFreeze

Enable or Disable Non-Correctable Errors In FlexCAN Access Put Device In Freeze Mode.

bool enableTransceiverDelayMeasure

Enable or Disable the transceiver delay measurement, when it is enabled, then the secondary sample point position is determined by the sum of the transceiver delay measurement plus the enhanced TDC offset.

bool enableRemoteRequestFrameStored

true: Store Remote Request Frame in the same fashion of data frame. false: Generate an automatic Remote Response Frame.

flexcan_endianness_t payloadEndianness

Selects the byte order for the payload of transmit and receive frames, see flexcan_endianness_t.

bool enableExternalTimeTick

true: External time tick clocks the free-running timer. false: FlexCAN bit clock clocks the free-running timer.

flexcan_MB_timestamp_base_t captureTimeBase

Timebase of message buffer 16-bit TIME_STAMP field.

flexcan_capture_point_t capturePoint

Point in time when 32-bit timebase is captured during CAN frame.

struct __flexcan__rx__mb__config

#include <fsl_flexcan.h> FlexCAN Receive Message Buffer configuration structure.

This structure is used as the parameter of FLEXCAN_SetRxMbConfig() function. The FLEXCAN_SetRxMbConfig() function is used to configure FlexCAN Receive Message Buffer. The function abort previous receiving process, clean the Message Buffer and activate the Rx Message Buffer using given Message Buffer setting.

Public Members

uint32_t id

CAN Message Buffer Frame Identifier, should be set using FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.

flexcan_frame_format_t format

CAN Frame Identifier format(Standard of Extend).

flexcan_frame_type_t type

CAN Frame Type(Data or Remote for classical CAN only).

struct __flexcan__pn__config

#include <fsl_flexcan.h> FlexCAN Pretended Networking configuration structure.

This structure is used as the parameter of FLEXCAN_SetPNConfig() function. The FLEXCAN_SetPNConfig() function is used to configure FlexCAN Networking work mode.

Public Members

bool enableTimeout

Enable or Disable timeout event trigger wakeup.

uint16_t timeoutValue

The timeout value that generates a wakeup event, the counter timer is incremented based on 64 times the CAN Bit Time unit.

bool enableMatch

Enable or Disable match event trigger wakeup.

flexcan_pn_match_source_t matchSrc

Selects the match source (ID and/or data match) to trigger wakeup.

uint8_t matchNum

The number of times a given message must match the predefined ID and/or data before generating a wakeup event, range in 0x1 ~ 0xFF.

flexcan_pn_match_mode_t idMatchMode

The ID match type.

flexcan_pn_match_mode_t dataMatchMode

The data match type.

uint32_t idLower
The ID target values 1 which used either for ID match "equal to", "smaller than", "greater than" comparisons, or as the lower limit value in ID match "range detection".

uint32_t idUpper
The ID target values 2 which used only as the upper limit value in ID match "range detection" or used to store the ID mask in "equal to".

uint8_t lengthLower
The lower limit for length of data bytes which used only in data match "range detection". Range in 0x0 ~ 0x8.

uint8_t lengthUpper
The upper limit for length of data bytes which used only in data match "range detection". Range in 0x0 ~ 0x8.

struct __flexcan_rx_fifo_config
#include <fsl_flexcan.h> FlexCAN Legacy Rx FIFO configuration structure.

Public Members

uint32_t *idFilterTable
Pointer to the FlexCAN Legacy Rx FIFO identifier filter table.

uint8_t idFilterNum
The FlexCAN Legacy Rx FIFO Filter elements quantity.

flexcan_rx_fifo_filter_type_t idFilterType
The FlexCAN Legacy Rx FIFO Filter type.

flexcan_rx_fifo_priority_t priority
The FlexCAN Legacy Rx FIFO receive priority.

struct __flexcan_enhanced_rx_fifo_std_id_filter
#include <fsl_flexcan.h> FlexCAN Enhanced Rx FIFO Standard ID filter element structure.

Public Members

uint32_t filterType
FlexCAN internal Free-Running Counter Time Stamp.

uint32_t rtr1
CAN FD frame data length code (DLC), range see _flexcan_fd_frame_length, When the length <= 8, it equal to the data length, otherwise the number of valid frame data is not equal to the length value. user can use DLC_LENGTH_DECODE(length) macro to get the number of valid data bytes.

uint32_t std1
CAN Frame Type(DATA or REMOTE).

uint32_t rtr2
CAN Frame Identifier(STD or EXT format).

uint32_t std2
Substitute Remote request.

struct __flexcan_enhanced_rx_fifo_ext_id_filter
#include <fsl_flexcan.h> FlexCAN Enhanced Rx FIFO Extended ID filter element structure.

Public Members

uint32_t filterType

FlexCAN internal Free-Running Counter Time Stamp.

uint32_t rtr1

CAN FD frame data length code (DLC), range see _flexcan_fd_frame_length, When the length <= 8, it equal to the data length, otherwise the number of valid frame data is not equal to the length value. user can use DLC_LENGTH_DECODE(length) macro to get the number of valid data bytes.

uint32_t std1

CAN Frame Type(DATA or REMOTE).

uint32_t rtr2

CAN Frame Identifier(STD or EXT format).

uint32_t std2

Substitute Remote request.

struct __flexcan__enhanced__rx__fifo__config

#include <fsl_flexcan.h> FlexCAN Enhanced Rx FIFO configuration structure.

Public Members

uint32_t *idFilterTable

Pointer to the FlexCAN Enhanced Rx FIFO identifier filter table, each table member occupies 32 bit word, table size should be equal to idFilterNum. There are two types of Enhanced Rx FIFO filter elements that can be stored in table : extended-ID filter element (1 word, occupie 1 table members) and standard-ID filter element (2 words, occupies 2 table members), the extended-ID filter element needs to be placed in front of the table.

uint8_t idFilterPairNum

idFilterPairNum is the Enhanced Rx FIFO identifier filter element pair numbers, each pair of filter elements occupies 2 words and can consist of one extended ID filter element or two standard ID filter elements.

uint8_t extendIdFilterNum

The number of extended ID filter element items in the FlexCAN enhanced Rx FIFO identifier filter table, each extended-ID filter element occupies 2 words, extendIdFilterNum need less than or equal to idFilterPairNum.

uint8_t fifoWatermark

(fifoWatermark + 1) is the minimum number of CAN messages stored in the Enhanced RX FIFO which can trigger FIFO watermark interrupt or a DMA request.

flexcan_efifo_dma_per_read_length_t dmaPerReadLength

Define the length of each read of the Enhanced RX FIFO element by the DAM, see _flexcan_fd_frame_length.

flexcan_rx_fifo_priority_t priority

The FlexCAN Enhanced Rx FIFO receive priority.

struct __flexcan__mb__transfer

#include <fsl_flexcan.h> FlexCAN Message Buffer transfer.

Public Members

flexcan_frame_t *frame
The buffer of CAN Message to be transfer.

uint8_t mbIdx
The index of Message buffer used to transfer Message.

struct __flexcan_fifo_transfer
#include <fsl_flexcan.h> FlexCAN Rx FIFO transfer.

Public Members

flexcan_fd_frame_t *framefd
The buffer of CAN Message to be received from Enhanced Rx FIFO.

flexcan_frame_t *frame
The buffer of CAN Message to be received from Legacy Rx FIFO.

size_t frameNum
Number of CAN Message need to be received from Legacy or Ehanced Rx FIFO.

struct __flexcan_handle
#include <fsl_flexcan.h> FlexCAN handle structure.

Public Members

flexcan_transfer_callback_t callback
Callback function.

void *userData
FlexCAN callback function parameter.

flexcan_frame_t *volatile mbFrameBuf[CAN_WORD1_COUNT]
The buffer for received CAN data from Message Buffers.

flexcan_fd_frame_t *volatile mbFDFrameBuf[CAN_WORD1_COUNT]
The buffer for received CAN FD data from Message Buffers.

flexcan_frame_t *volatile rxFifoFrameBuf
The buffer for received CAN data from Legacy Rx FIFO.

flexcan_fd_frame_t *volatile rxFifoFDFrameBuf
The buffer for received CAN FD data from Ehanced Rx FIFO.

size_t rxFifoFrameNum
The number of CAN messages remaining to be received from Legacy or Ehanced Rx FIFO.

size_t rxFifoTransferTotalNum
Total CAN Message number need to be received from Legacy or Ehanced Rx FIFO.

volatile uint8_t mbState[CAN_WORD1_COUNT]
Message Buffer transfer state.

volatile uint8_t rxFifoState
Rx FIFO transfer state.

volatile uint32_t timestamp[CAN_WORD1_COUNT]
Mailbox transfer timestamp.

struct byteStatus

Public Members

bool byteIsRead
: The byte n (0~3) was read or not. The type of error and which bit in byte (n) is affected by the error.

struct ___unnamed14___

Public Members

uint32_t timestamp
: FlexCAN internal Free-Running Counter Time Stamp.

uint32_t length
: CAN frame data length in bytes (Range: 0~8).

uint32_t type
: CAN Frame Type(DATA or REMOTE).

uint32_t format
: CAN Frame Identifier(STD or EXT format).

uint32_t ___pad0___
: Reserved.

uint32_t idhit
: CAN Rx FIFO filter hit id(This value is only used in Rx FIFO receive mode).

struct ___unnamed16___

Public Members

uint32_t id
: CAN Frame Identifier, should be set using FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.

uint32_t ___pad0___
: Reserved.

union ___unnamed18___

Public Members

struct __flexcan_frame

struct __flexcan_frame

struct ___unnamed20___

Public Members

uint32_t dataWord0
: CAN Frame payload word0.

uint32_t dataWord1
: CAN Frame payload word1.

struct ___unnamed22___

Public Members

uint8_t dataByte3
CAN Frame payload byte3.

uint8_t dataByte2
CAN Frame payload byte2.

uint8_t dataByte1
CAN Frame payload byte1.

uint8_t dataByte0
CAN Frame payload byte0.

uint8_t dataByte7
CAN Frame payload byte7.

uint8_t dataByte6
CAN Frame payload byte6.

uint8_t dataByte5
CAN Frame payload byte5.

uint8_t dataByte4
CAN Frame payload byte4.

struct ___unnamed24___

Public Members

uint32_t timestamp
FlexCAN internal Free-Running Counter Time Stamp.

uint32_t length
CAN FD frame data length code (DLC), range see _flexcan_fd_frame_length, When the length <= 8, it equal to the data length, otherwise the number of valid frame data is not equal to the length value. user can use DLC_LENGTH_DECODE(length) macro to get the number of valid data bytes.

uint32_t type
CAN Frame Type(DATA only).

uint32_t format
CAN Frame Identifier(STD or EXT format).

uint32_t srr
Substitute Remote request.

uint32_t esi
Error State Indicator.

uint32_t brs
Bit Rate Switch.

uint32_t edl
Extended Data Length.

struct ___unnamed26___

Public Members

uint32_t id
: CAN Frame Identifier, should be set using FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.

uint32_t ___pad0___
: Reserved.

union ___unnamed28___

Public Members

struct __flexcan__fd__frame

struct __flexcan__fd__frame

struct ___unnamed30___

Public Members

uint32_t dataWord[16]
: CAN FD Frame payload, 16 double word maximum.

struct ___unnamed32___

Public Members

uint8_t dataByte3
: CAN Frame payload byte3.

uint8_t dataByte2
: CAN Frame payload byte2.

uint8_t dataByte1
: CAN Frame payload byte1.

uint8_t dataByte0
: CAN Frame payload byte0.

uint8_t dataByte7
: CAN Frame payload byte7.

uint8_t dataByte6
: CAN Frame payload byte6.

uint8_t dataByte5
: CAN Frame payload byte5.

uint8_t dataByte4
: CAN Frame payload byte4.

union ___unnamed34___

Public Members

struct __flexcan_config

struct __flexcan_config

struct ___unnamed36___

Public Members

uint32_t baudRate
FlexCAN bit rate in bps, for classical CAN or CANFD nominal phase.

uint32_t baudRateFD
FlexCAN FD bit rate in bps, for CANFD data phase.

struct ___unnamed38___

Public Members

uint32_t bitRate
FlexCAN bit rate in bps, for classical CAN or CANFD nominal phase.

uint32_t bitRateFD
FlexCAN FD bit rate in bps, for CANFD data phase.

union ___unnamed40___

Public Members

struct __flexcan_pn_config

< The data target values 1 which used either for data match “equal to”, “smaller than”, “greater than” comparisons, or as the lower limit value in data match “range

detection”.

struct __flexcan_pn_config

struct ___unnamed44___

< The data target values 1 which used either for data match “equal to”, “smaller than”, “greater than” comparisons, or as the lower limit value in data match “range

detection”.

Public Members

uint32_t lowerWord0
CAN Frame payload word0.

uint32_t lowerWord1
CAN Frame payload word1.

struct ___unnamed46___

Public Members

uint8_t lowerByte3
CAN Frame payload byte3.

uint8_t lowerByte2
CAN Frame payload byte2.

uint8_t lowerByte1
CAN Frame payload byte1.

uint8_t lowerByte0
CAN Frame payload byte0.

uint8_t lowerByte7
CAN Frame payload byte7.

uint8_t lowerByte6
CAN Frame payload byte6.

uint8_t lowerByte5
CAN Frame payload byte5.

uint8_t lowerByte4
CAN Frame payload byte4.

union ___unnamed42___

Public Members

struct __flexcan__pn__config

< The data target values 2 which used only as the upper limit value in data match "range

detection" or used to store the data mask in "equal to".

struct __flexcan__pn__config

struct ___unnamed48___

< The data target values 2 which used only as the upper limit value in data match "range

detection" or used to store the data mask in "equal to".

Public Members

uint32_t upperWord0
CAN Frame payload word0.

uint32_t upperWord1
CAN Frame payload word1.

struct ___unnamed50___

Public Members

uint8_t upperByte3
CAN Frame payload byte3.

uint8_t upperByte2
CAN Frame payload byte2.

uint8_t upperByte1
CAN Frame payload byte1.

uint8_t upperByte0
CAN Frame payload byte0.

uint8_t upperByte7
CAN Frame payload byte7.

uint8_t upperByte6
CAN Frame payload byte6.

uint8_t upperByte5
CAN Frame payload byte5.

uint8_t upperByte4
CAN Frame payload byte4.

2.13 FlexCAN eDMA Driver

void FLEXCAN_TransferCreateHandleEDMA(CAN_Type *base, *flexcan_edma_handle_t* *handle, *flexcan_edma_transfer_callback_t* callback, void *userData, *edma_handle_t* *rxFifoEdmaHandle)

Initializes the FlexCAN handle, which is used in transactional functions.

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.
- callback – The callback function.
- userData – The parameter of the callback function.
- rxFifoEdmaHandle – User-requested DMA handle for Rx FIFO DMA transfer.

void FLEXCAN_PrepareTransfConfiguration(CAN_Type *base, *flexcan_fifo_transfer_t* *pFifoXfer, *edma_transfer_config_t* *pEdmaConfig)

Prepares the eDMA transfer configuration for FLEXCAN Legacy RX FIFO.

This function prepares the eDMA transfer configuration structure according to FLEXCAN Legacy RX FIFO.

Parameters

- base – FlexCAN peripheral base address.
- pFifoXfer – FlexCAN Rx FIFO EDMA transfer structure, see flexcan_fifo_transfer_t.
- pEdmaConfig – The user configuration structure of type edma_transfer_t.

status_t FLEXCAN_StartTransferDatafromRxFIFO(CAN_Type *base, *flexcan_edma_handle_t* *handle, *edma_transfer_config_t* *pEdmaConfig)

Start Transfer Data from the FLEXCAN Legacy Rx FIFO using eDMA.

This function to Update edma transfer confiugration and Start eDMA transfer

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.
- pEdmaConfig – The user configuration structure of type edma_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_FLEXCAN_RxFifoBusy – Previous transfer ongoing.

status_t FLEXCAN_TransferReceiveFifoEDMA(CAN_Type *base, *flexcan_edma_handle_t* *handle, *flexcan_fifo_transfer_t* *pFifoXfer)

Receives the CAN Message from the Legacy Rx FIFO using eDMA.

This function receives the CAN Message using eDMA. This is a non-blocking function, which returns right away. After the CAN Message is received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.
- pFifoXfer – FlexCAN Rx FIFO EDMA transfer structure, see flexcan_fifo_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_FLEXCAN_RxFifoBusy – Previous transfer ongoing.

status_t FLEXCAN_TransferGetReceiveFifoCountEMDA(CAN_Type *base, *flexcan_edma_handle_t* *handle, size_t *count)

Gets the Legacy Rx Fifo transfer status during a interrupt non-blocking receive.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- count – Number of CAN messages receive so far by the non-blocking transaction.

Return values

- kStatus_InvalidArgument – count is Invalid.
- kStatus_Success – Successfully return the count.

void FLEXCAN_TransferAbortReceiveFifoEDMA(CAN_Type *base, *flexcan_edma_handle_t* *handle)

Aborts the receive Legacy/Enhanced Rx FIFO process which used eDMA.

This function aborts the receive Legacy/Enhanced Rx FIFO process which used eDMA.

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.

status_t FLEXCAN_TransferReceiveEnhancedFifoEDMA(CAN_Type *base, *flexcan_edma_handle_t* *handle, *flexcan_fifo_transfer_t* *pFifoXfer)

Receives the CAN FD Message from the Enhanced Rx FIFO using eDMA.

This function receives the CAN FD Message using eDMA. This is a non-blocking function, which returns right away. After the CAN Message is received, the receive callback function is called.

Parameters

- base – FlexCAN peripheral base address.
- handle – Pointer to flexcan_edma_handle_t structure.
- pFifoXfer – FlexCAN Rx FIFO EDMA transfer structure, see flexcan_fifo_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_FLEXCAN_RxFifoBusy – Previous transfer ongoing.

static inline *status_t* FLEXCAN_TransferGetReceiveEnhancedFifoCountEMDA(CAN_Type *base, *flexcan_edma_handle_t* *handle, size_t *count)

Gets the Enhanced Rx Fifo transfer status during a interrupt non-blocking receive.

Parameters

- base – FlexCAN peripheral base address.
- handle – FlexCAN handle pointer.
- count – Number of CAN messages receive so far by the non-blocking transaction.

Return values

- kStatus_InvalidArgument – count is Invalid.
- kStatus_Success – Successfully return the count.

FSL_FLEXCAN_EDMA_DRIVER_VERSION

FlexCAN EDMA driver version.

typedef struct *_flexcan_edma_handle* flexcan_edma_handle_t

typedef void (*flexcan_edma_transfer_callback_t)(CAN_Type *base, *flexcan_edma_handle_t* *handle, *status_t* status, void *userData)

FlexCAN transfer callback function.

struct _flexcan_edma_handle

#include <fsl_flexcan_edma.h> FlexCAN eDMA handle.

Public Members

flexcan_edma_transfer_callback_t callback

Callback function.

void *userData

FlexCAN callback function parameter.

edma_handle_t *rxFifoEdmaHandle

The EDMA handler for Rx FIFO.

volatile uint8_t rxFifoState
Rx FIFO transfer state.

size_t frameNum
The number of messages that need to be received.

flexcan_fd_frame_t *framefd
Point to the buffer of CAN Message to be received from Enhanced Rx FIFO.

2.14 FlexIO: FlexIO Driver

2.15 FlexIO Driver

void FLEXIO_GetDefaultConfig(*flexio_config_t* *userConfig)
Gets the default configuration to configure the FlexIO module. The configuration can used directly to call the FLEXIO_Configure().

Example:

```
flexio_config_t config;
FLEXIO_GetDefaultConfig(&config);
```

Parameters

- userConfig – pointer to flexio_config_t structure

void FLEXIO_Init(FLEXIO_Type *base, const *flexio_config_t* *userConfig)
Configures the FlexIO with a FlexIO configuration. The configuration structure can be filled by the user or be set with default values by FLEXIO_GetDefaultConfig().

Example

```
flexio_config_t config = {
.enableFlexio = true,
.enableInDoze = false,
.enableInDebug = true,
.enableFastAccess = false
};
FLEXIO_Configure(base, &config);
```

Parameters

- base – FlexIO peripheral base address
- userConfig – pointer to flexio_config_t structure

void FLEXIO_Deinit(FLEXIO_Type *base)
Gates the FlexIO clock. Call this API to stop the FlexIO clock.

Note: After calling this API, call the FLEXO_Init to use the FlexIO module.

Parameters

- base – FlexIO peripheral base address

uint32_t FLEXIO_GetInstance(FLEXIO_Type *base)

Get instance number for FLEXIO module.

Parameters

- base – FLEXIO peripheral base address.

void FLEXIO_Reset(FLEXIO_Type *base)

Resets the FlexIO module.

Parameters

- base – FlexIO peripheral base address

static inline void FLEXIO_Enable(FLEXIO_Type *base, bool enable)

Enables the FlexIO module operation.

Parameters

- base – FlexIO peripheral base address
- enable – true to enable, false to disable.

static inline uint32_t FLEXIO_ReadPinInput(FLEXIO_Type *base)

Reads the input data on each of the FlexIO pins.

Parameters

- base – FlexIO peripheral base address

Returns

FlexIO pin input data

static inline uint8_t FLEXIO_GetShifterState(FLEXIO_Type *base)

Gets the current state pointer for state mode use.

Parameters

- base – FlexIO peripheral base address

Returns

current State pointer

void FLEXIO_SetShifterConfig(FLEXIO_Type *base, uint8_t index, const *flexio_shifter_config_t* *shifterConfig)

Configures the shifter with the shifter configuration. The configuration structure covers both the SHIFTCTL and SHIFTCFG registers. To configure the shifter to the proper mode, select which timer controls the shifter to shift, whether to generate start bit/stop bit, and the polarity of start bit and stop bit.

Example

```
flexio_shifter_config_t config = {
.timerSelect = 0,
.timerPolarity = kFLEXIO_ShifterTimerPolarityOnPositive,
.pinConfig = kFLEXIO_PinConfigOpenDrainOrBidirection,
.pinPolarity = kFLEXIO_PinActiveLow,
.shifterMode = kFLEXIO_ShifterModeTransmit,
.inputSource = kFLEXIO_ShifterInputFromPin,
.shifterStop = kFLEXIO_ShifterStopBitHigh,
.shifterStart = kFLEXIO_ShifterStartBitLow
};
FLEXIO_SetShifterConfig(base, &config);
```

Parameters

- base – FlexIO peripheral base address

- index – Shifter index
- shifterConfig – Pointer to flexio_shifter_config_t structure

void FLEXIO_SetTimerConfig(FLEXIO_Type *base, uint8_t index, const *flexio_timer_config_t* *timerConfig)

Configures the timer with the timer configuration. The configuration structure covers both the TIMCTL and TIMCFG registers. To configure the timer to the proper mode, select trigger source for timer and the timer pin output and the timing for timer.

Example

```
flexio_timer_config_t config = {
.triggerSelect = FLEXIO_TIMER_TRIGGER_SEL_SHIFTnSTAT(0),
.triggerPolarity = kFLEXIO_TimerTriggerPolarityActiveLow,
.triggerSource = kFLEXIO_TimerTriggerSourceInternal,
.pinConfig = kFLEXIO_PinConfigOpenDrainOrBidirection,
.pinSelect = 0,
.pinPolarity = kFLEXIO_PinActiveHigh,
.timerMode = kFLEXIO_TimerModeDual8BitBaudBit,
.timerOutput = kFLEXIO_TimerOutputZeroNotAffectedByReset,
.timerDecrement = kFLEXIO_TimerDecSrcOnFlexIOClockShiftTimerOutput,
.timerReset = kFLEXIO_TimerResetOnTimerPinEqualToTimerOutput,
.timerDisable = kFLEXIO_TimerDisableOnTimerCompare,
.timerEnable = kFLEXIO_TimerEnableOnTriggerHigh,
.timerStop = kFLEXIO_TimerStopBitEnableOnTimerDisable,
.timerStart = kFLEXIO_TimerStartBitEnabled
};
FLEXIO_SetTimerConfig(base, &config);
```

Parameters

- base – FlexIO peripheral base address
- index – Timer index
- timerConfig – Pointer to the flexio_timer_config_t structure

static inline void FLEXIO_SetClockMode(FLEXIO_Type *base, uint8_t index, *flexio_timer_decrement_source_t* clocksource)

This function set the value of the prescaler on flexio channels.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.
- index – Timer index
- clocksource – Set clock value

static inline void FLEXIO_EnableShifterStatusInterrupts(FLEXIO_Type *base, uint32_t mask)

Enables the shifter status interrupt. The interrupt generates when the corresponding SSF is set.

Note: For multiple shifter status interrupt enable, for example, two shifter status enable, can calculate the mask by using ((1 « shifter index0) | (1 « shifter index1))

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by (1 « shifter index)

static inline void FLEXIO_DisableShifterStatusInterrupts(FLEXIO_Type *base, uint32_t mask)

Disables the shifter status interrupt. The interrupt won't generate when the corresponding SSF is set.

Note: For multiple shifter status interrupt enable, for example, two shifter status enable, can calculate the mask by using ((1 « shifter index0) | (1 « shifter index1))

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by (1 « shifter index)

static inline void FLEXIO_EnableShifterErrorInterrupts(FLEXIO_Type *base, uint32_t mask)

Enables the shifter error interrupt. The interrupt generates when the corresponding SEF is set.

Note: For multiple shifter error interrupt enable, for example, two shifter error enable, can calculate the mask by using ((1 « shifter index0) | (1 « shifter index1))

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by (1 « shifter index)

static inline void FLEXIO_DisableShifterErrorInterrupts(FLEXIO_Type *base, uint32_t mask)

Disables the shifter error interrupt. The interrupt won't generate when the corresponding SEF is set.

Note: For multiple shifter error interrupt enable, for example, two shifter error enable, can calculate the mask by using ((1 « shifter index0) | (1 « shifter index1))

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by (1 « shifter index)

static inline void FLEXIO_EnableTimerStatusInterrupts(FLEXIO_Type *base, uint32_t mask)

Enables the timer status interrupt. The interrupt generates when the corresponding SSF is set.

Note: For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using ((1 « timer index0) | (1 « timer index1))

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by (1 « timer index)

static inline void FLEXIO_DisableTimerStatusInterrupts(FLEXIO_Type *base, uint32_t mask)

Disables the timer status interrupt. The interrupt won't generate when the corresponding SSF is set.

Note: For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using ((1 « timer index0) | (1 « timer index1))

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by (1 « timer index)

static inline uint32_t FLEXIO_GetShifterStatusFlags(FLEXIO_Type *base)

Gets the shifter status flags.

Parameters

- base – FlexIO peripheral base address

Returns

Shifter status flags

static inline void FLEXIO_ClearShifterStatusFlags(FLEXIO_Type *base, uint32_t mask)

Clears the shifter status flags.

Note: For clearing multiple shifter status flags, for example, two shifter status flags, can calculate the mask by using ((1 « shifter index0) | (1 « shifter index1))

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by (1 « shifter index)

static inline uint32_t FLEXIO_GetShifterErrorFlags(FLEXIO_Type *base)

Gets the shifter error flags.

Parameters

- base – FlexIO peripheral base address

Returns

Shifter error flags

static inline void FLEXIO_ClearShifterErrorFlags(FLEXIO_Type *base, uint32_t mask)

Clears the shifter error flags.

Note: For clearing multiple shifter error flags, for example, two shifter error flags, can calculate the mask by using ((1 « shifter index0) | (1 « shifter index1))

Parameters

- base – FlexIO peripheral base address
- mask – The shifter error mask which can be calculated by (1 « shifter index)

static inline uint32_t FLEXIO_GetTimerStatusFlags(FLEXIO_Type *base)

Gets the timer status flags.

Parameters

- base – FlexIO peripheral base address

Returns

Timer status flags

static inline void FLEXIO_ClearTimerStatusFlags(FLEXIO_Type *base, uint32_t mask)

Clears the timer status flags.

Note: For clearing multiple timer status flags, for example, two timer status flags, can calculate the mask by using ((1 « timer index0) | (1 « timer index1))

Parameters

- base – FlexIO peripheral base address
- mask – The timer status mask which can be calculated by (1 « timer index)

static inline void FLEXIO_EnableShifterStatusDMA(FLEXIO_Type *base, uint32_t mask, bool enable)

Enables/disables the shifter status DMA. The DMA request generates when the corresponding SSF is set.

Note: For multiple shifter status DMA enables, for example, calculate the mask by using ((1 « shifter index0) | (1 « shifter index1))

Parameters

- base – FlexIO peripheral base address
- mask – The shifter status mask which can be calculated by (1 « shifter index)
- enable – True to enable, false to disable.

uint32_t FLEXIO_GetShifterBufferAddress(FLEXIO_Type *base, *flexio_shifter_buffer_type_t* type, uint8_t index)

Gets the shifter buffer address for the DMA transfer usage.

Parameters

- base – FlexIO peripheral base address
- type – Shifter type of flexio_shifter_buffer_type_t
- index – Shifter index

Returns

Corresponding shifter buffer index

status_t FLEXIO_RegisterHandleIRQ(void *base, void *handle, *flexio_isr_t* isr)

Registers the handle and the interrupt handler for the FlexIO-simulated peripheral.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.
- handle – Pointer to the handler for FlexIO simulated peripheral.
- isr – FlexIO simulated peripheral interrupt handler.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

status_t FLEXIO_UnregisterHandleIRQ(void *base)

Unregisters the handle and the interrupt handler for the FlexIO-simulated peripheral.

Parameters

- base – Pointer to the FlexIO simulated peripheral type.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

static inline void FLEXIO_ClearPortOutput(FLEXIO_Type *base, uint32_t mask)

Sets the output level of the multiple FLEXIO pins to the logic 0.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

static inline void FLEXIO_SetPortOutput(FLEXIO_Type *base, uint32_t mask)

Sets the output level of the multiple FLEXIO pins to the logic 1.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

static inline void FLEXIO_TogglePortOutput(FLEXIO_Type *base, uint32_t mask)

Reverses the current output logic of the multiple FLEXIO pins.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

static inline void FLEXIO_PinWrite(FLEXIO_Type *base, uint32_t pin, uint8_t output)

Sets the output level of the FLEXIO pins to the logic 1 or 0.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.
- output – FLEXIO pin output logic level.
  - 0: corresponding pin output low-logic level.
  - 1: corresponding pin output high-logic level.

static inline void FLEXIO_EnablePinOutput(FLEXIO_Type *base, uint32_t pin)

Enables the FLEXIO output pin function.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.

static inline uint32_t FLEXIO_PinRead(FLEXIO_Type *base, uint32_t pin)

Reads the current input value of the FLEXIO pin.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.

Return values

FLEXIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

static inline uint32_t FLEXIO_GetPinStatus(FLEXIO_Type *base, uint32_t pin)

Gets the FLEXIO input pin status.

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.

Return values

FLEXIO – port input status

- 0: corresponding pin input capture no status.
- 1: corresponding pin input capture rising or falling edge.

static inline void FLEXIO_SetPinLevel(FLEXIO_Type *base, uint8_t pin, bool level)

Sets the FLEXIO output pin level.

Parameters

- base – FlexIO peripheral base address
- pin – FlexIO pin number.
- level – FlexIO output pin level to set, can be either 0 or 1.

static inline bool FLEXIO_GetPinOverride(const FLEXIO_Type *const base, uint8_t pin)

Gets the enabled status of a FLEXIO output pin.

Parameters

- base – FlexIO peripheral base address
- pin – FlexIO pin number.

Return values

FlexIO – port enabled status

- 0: corresponding output pin is in disabled state.
- 1: corresponding output pin is in enabled state.

static inline void FLEXIO_ConfigPinOverride(FLEXIO_Type *base, uint8_t pin, bool enabled)

Enables or disables a FLEXIO output pin.

Parameters

- base – FlexIO peripheral base address
- pin – Flexio pin number.
- enabled – Enable or disable the FlexIO pin.

static inline void FLEXIO_ClearPortStatus(FLEXIO_Type *base, uint32_t mask)

Clears the multiple FLEXIO input pins status.

Parameters

- base – FlexIO peripheral base address
- mask – FLEXIO pin number mask

FSL_FLEXIO_DRIVER_VERSION

FlexIO driver version.

enum _flexio_timer_trigger_polarity

Define time of timer trigger polarity.

Values:

enumerator kFLEXIO_TimerTriggerPolarityActiveHigh

Active high.

enumerator kFLEXIO_TimerTriggerPolarityActiveLow

Active low.

enum _flexio_timer_trigger_source

Define type of timer trigger source.

Values:

enumerator kFLEXIO_TimerTriggerSourceExternal

External trigger selected.

enumerator kFLEXIO_TimerTriggerSourceInternal

Internal trigger selected.

enum _flexio_pin_config

Define type of timer/shifter pin configuration.

Values:

enumerator kFLEXIO_PinConfigOutputDisabled

Pin output disabled.

enumerator kFLEXIO_PinConfigOpenDrainOrBidirection

Pin open drain or bidirectional output enable.

enumerator kFLEXIO_PinConfigBidirectionOutputData

Pin bidirectional output data.

enumerator kFLEXIO_PinConfigOutput

Pin output.

enum _flexio_pin_polarity

Definition of pin polarity.

Values:

enumerator kFLEXIO_PinActiveHigh

Active high.

enumerator kFLEXIO_PinActiveLow

Active low.

enum _flexio_timer_mode

Define type of timer work mode.

Values:

enumerator kFLEXIO_TimerModeDisabled
Timer Disabled.

enumerator kFLEXIO_TimerModeDual8BitBaudBit
Dual 8-bit counters baud/bit mode.

enumerator kFLEXIO_TimerModeDual8BitPWM
Dual 8-bit counters PWM mode.

enumerator kFLEXIO_TimerModeSingle16Bit
Single 16-bit counter mode.

enumerator kFLEXIO_TimerModeDual8BitPWMLow
Dual 8-bit counters PWM Low mode.

enum _flexio_timer_output
Define type of timer initial output or timer reset condition.

Values:

enumerator kFLEXIO_TimerOutputOneNotAffectedByReset
Logic one when enabled and is not affected by timer reset.

enumerator kFLEXIO_TimerOutputZeroNotAffectedByReset
Logic zero when enabled and is not affected by timer reset.

enumerator kFLEXIO_TimerOutputOneAffectedByReset
Logic one when enabled and on timer reset.

enumerator kFLEXIO_TimerOutputZeroAffectedByReset
Logic zero when enabled and on timer reset.

enum _flexio_timer_decrement_source
Define type of timer decrement.

Values:

enumerator kFLEXIO_TimerDecSrcOnFlexIOClockShiftTimerOutput
Decrement counter on FlexIO clock, Shift clock equals Timer output.

enumerator kFLEXIO_TimerDecSrcOnTriggerInputShiftTimerOutput
Decrement counter on Trigger input (both edges), Shift clock equals Timer output.

enumerator kFLEXIO_TimerDecSrcOnPinInputShiftPinInput
Decrement counter on Pin input (both edges), Shift clock equals Pin input.

enumerator kFLEXIO_TimerDecSrcOnTriggerInputShiftTriggerInput
Decrement counter on Trigger input (both edges), Shift clock equals Trigger input.

enum _flexio_timer_reset_condition
Define type of timer reset condition.

Values:

enumerator kFLEXIO_TimerResetNever
Timer never reset.

enumerator kFLEXIO_TimerResetOnTimerPinEqualToTimerOutput
Timer reset on Timer Pin equal to Timer Output.

enumerator kFLEXIO_TimerResetOnTimerTriggerEqualToTimerOutput
Timer reset on Timer Trigger equal to Timer Output.

enumerator kFLEXIO_TimerResetOnTimerPinRisingEdge
Timer reset on Timer Pin rising edge.

enumerator kFLEXIO_TimerResetOnTimerTriggerRisingEdge
Timer reset on Trigger rising edge.

enumerator kFLEXIO_TimerResetOnTimerTriggerBothEdge
Timer reset on Trigger rising or falling edge.

enum _flexio_timer_disable_condition
Define type of timer disable condition.

Values:

enumerator kFLEXIO_TimerDisableNever
Timer never disabled.

enumerator kFLEXIO_TimerDisableOnPreTimerDisable
Timer disabled on Timer N-1 disable.

enumerator kFLEXIO_TimerDisableOnTimerCompare
Timer disabled on Timer compare.

enumerator kFLEXIO_TimerDisableOnTimerCompareTriggerLow
Timer disabled on Timer compare and Trigger Low.

enumerator kFLEXIO_TimerDisableOnPinBothEdge
Timer disabled on Pin rising or falling edge.

enumerator kFLEXIO_TimerDisableOnPinBothEdgeTriggerHigh
Timer disabled on Pin rising or falling edge provided Trigger is high.

enumerator kFLEXIO_TimerDisableOnTriggerFallingEdge
Timer disabled on Trigger falling edge.

enum _flexio_timer_enable_condition
Define type of timer enable condition.

Values:

enumerator kFLEXIO_TimerEnabledAlways
Timer always enabled.

enumerator kFLEXIO_TimerEnableOnPrevTimerEnable
Timer enabled on Timer N-1 enable.

enumerator kFLEXIO_TimerEnableOnTriggerHigh
Timer enabled on Trigger high.

enumerator kFLEXIO_TimerEnableOnTriggerHighPinHigh
Timer enabled on Trigger high and Pin high.

enumerator kFLEXIO_TimerEnableOnPinRisingEdge
Timer enabled on Pin rising edge.

enumerator kFLEXIO_TimerEnableOnPinRisingEdgeTriggerHigh
Timer enabled on Pin rising edge and Trigger high.

enumerator kFLEXIO_TimerEnableOnTriggerRisingEdge
Timer enabled on Trigger rising edge.

enumerator kFLEXIO_TimerEnableOnTriggerBothEdge
Timer enabled on Trigger rising or falling edge.

enum _flexio_timer_stop_bit_condition

Define type of timer stop bit generate condition.

Values:

enumerator kFLEXIO_TimerStopBitDisabled

Stop bit disabled.

enumerator kFLEXIO_TimerStopBitEnableOnTimerCompare

Stop bit is enabled on timer compare.

enumerator kFLEXIO_TimerStopBitEnableOnTimerDisable

Stop bit is enabled on timer disable.

enumerator kFLEXIO_TimerStopBitEnableOnTimerCompareDisable

Stop bit is enabled on timer compare and timer disable.

enum _flexio_timer_start_bit_condition

Define type of timer start bit generate condition.

Values:

enumerator kFLEXIO_TimerStartBitDisabled

Start bit disabled.

enumerator kFLEXIO_TimerStartBitEnabled

Start bit enabled.

enum _flexio_timer_output_state

FlexIO as PWM channel output state.

Values:

enumerator kFLEXIO_PwmLow

The output state of PWM channel is low

enumerator kFLEXIO_PwmHigh

The output state of PWM channel is high

enum _flexio_shifter_timer_polarity

Define type of timer polarity for shifter control.

Values:

enumerator kFLEXIO_ShifterTimerPolarityOnPositive

Shift on positive edge of shift clock.

enumerator kFLEXIO_ShifterTimerPolarityOnNegitive

Shift on negative edge of shift clock.

enum _flexio_shifter_mode

Define type of shifter working mode.

Values:

enumerator kFLEXIO_ShifterDisabled

Shifter is disabled.

enumerator kFLEXIO_ShifterModeReceive

Receive mode.

enumerator kFLEXIO_ShifterModeTransmit

Transmit mode.

enumerator kFLEXIO_ShifterModeMatchStore
Match store mode.

enumerator kFLEXIO_ShifterModeMatchContinuous
Match continuous mode.

enumerator kFLEXIO_ShifterModeState
SHIFTBUF contents are used for storing programmable state attributes.

enumerator kFLEXIO_ShifterModeLogic
SHIFTBUF contents are used for implementing programmable logic look up table.

enum _flexio_shifter_input_source
Define type of shifter input source.

Values:

enumerator kFLEXIO_ShifterInputFromPin
Shifter input from pin.

enumerator kFLEXIO_ShifterInputFromNextShifterOutput
Shifter input from Shifter N+1.

enum _flexio_shifter_stop_bit
Define of STOP bit configuration.

Values:

enumerator kFLEXIO_ShifterStopBitDisable
Disable shifter stop bit.

enumerator kFLEXIO_ShifterStopBitLow
Set shifter stop bit to logic low level.

enumerator kFLEXIO_ShifterStopBitHigh
Set shifter stop bit to logic high level.

enum _flexio_shifter_start_bit
Define type of START bit configuration.

Values:

enumerator kFLEXIO_ShifterStartBitDisabledLoadDataOnEnable
Disable shifter start bit, transmitter loads data on enable.

enumerator kFLEXIO_ShifterStartBitDisabledLoadDataOnShift
Disable shifter start bit, transmitter loads data on first shift.

enumerator kFLEXIO_ShifterStartBitLow
Set shifter start bit to logic low level.

enumerator kFLEXIO_ShifterStartBitHigh
Set shifter start bit to logic high level.

enum _flexio_shifter_buffer_type
Define FlexIO shifter buffer type.

Values:

enumerator kFLEXIO_ShifterBuffer
Shifter Buffer N Register.

enumerator kFLEXIO_ShifterBufferBitSwapped
Shifter Buffer N Bit Byte Swapped Register.

enumerator kFLEXIO_ShifterBufferByteSwapped
Shifter Buffer N Byte Swapped Register.

enumerator kFLEXIO_ShifterBufferBitByteSwapped
Shifter Buffer N Bit Swapped Register.

enumerator kFLEXIO_ShifterBufferNibbleByteSwapped
Shifter Buffer N Nibble Byte Swapped Register.

enumerator kFLEXIO_ShifterBufferHalfWordSwapped
Shifter Buffer N Half Word Swapped Register.

enumerator kFLEXIO_ShifterBufferNibbleSwapped
Shifter Buffer N Nibble Swapped Register.

enum _flexio_gpio_direction
FLEXIO gpio direction definition.

Values:

enumerator kFLEXIO_DigitalInput
Set current pin as digital input

enumerator kFLEXIO_DigitalOutput
Set current pin as digital output

enum _flexio_pin_input_config
FLEXIO gpio input config.

Values:

enumerator kFLEXIO_InputInterruptDisabled
Interrupt request is disabled.

enumerator kFLEXIO_InputInterruptEnable
Interrupt request is enable.

enumerator kFLEXIO_FlagRisingEdgeEnable
Input pin flag on rising edge.

enumerator kFLEXIO_FlagFallingEdgeEnable
Input pin flag on falling edge.

typedef enum *_flexio_timer_trigger_polarity* flexio_timer_trigger_polarity_t
Define time of timer trigger polarity.

typedef enum *_flexio_timer_trigger_source* flexio_timer_trigger_source_t
Define type of timer trigger source.

typedef enum *_flexio_pin_config* flexio_pin_config_t
Define type of timer/shifter pin configuration.

typedef enum *_flexio_pin_polarity* flexio_pin_polarity_t
Definition of pin polarity.

typedef enum *_flexio_timer_mode* flexio_timer_mode_t
Define type of timer work mode.

typedef enum *_flexio_timer_output* flexio_timer_output_t
Define type of timer initial output or timer reset condition.

typedef enum *_flexio_timer_decrement_source* flexio_timer_decrement_source_t
Define type of timer decrement.

typedef enum *_flexio_timer_reset_condition* flexio_timer_reset_condition_t
Define type of timer reset condition.

typedef enum *_flexio_timer_disable_condition* flexio_timer_disable_condition_t
Define type of timer disable condition.

typedef enum *_flexio_timer_enable_condition* flexio_timer_enable_condition_t
Define type of timer enable condition.

typedef enum *_flexio_timer_stop_bit_condition* flexio_timer_stop_bit_condition_t
Define type of timer stop bit generate condition.

typedef enum *_flexio_timer_start_bit_condition* flexio_timer_start_bit_condition_t
Define type of timer start bit generate condition.

typedef enum *_flexio_timer_output_state* flexio_timer_output_state_t
FlexIO as PWM channel output state.

typedef enum *_flexio_shifter_timer_polarity* flexio_shifter_timer_polarity_t
Define type of timer polarity for shifter control.

typedef enum *_flexio_shifter_mode* flexio_shifter_mode_t
Define type of shifter working mode.

typedef enum *_flexio_shifter_input_source* flexio_shifter_input_source_t
Define type of shifter input source.

typedef enum *_flexio_shifter_stop_bit* flexio_shifter_stop_bit_t
Define of STOP bit configuration.

typedef enum *_flexio_shifter_start_bit* flexio_shifter_start_bit_t
Define type of START bit configuration.

typedef enum *_flexio_shifter_buffer_type* flexio_shifter_buffer_type_t
Define FlexIO shifter buffer type.

typedef struct *_flexio_config_* flexio_config_t
Define FlexIO user configuration structure.

typedef struct *_flexio_timer_config* flexio_timer_config_t
Define FlexIO timer configuration structure.

typedef struct *_flexio_shifter_config* flexio_shifter_config_t
Define FlexIO shifter configuration structure.

typedef enum *_flexio_gpio_direction* flexio_gpio_direction_t
FLEXIO gpio direction definition.

typedef enum *_flexio_pin_input_config* flexio_pin_input_config_t
FLEXIO gpio input config.

typedef struct *_flexio_gpio_config* flexio_gpio_config_t
The FLEXIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, use inputConfig param. If configured as an output pin, use outputLogic.

typedef void (*flexio_isr_t)(void *base, void *handle)
typedef for FlexIO simulated driver interrupt handler.

FLEXIO_Type *const s_flexioBases[]
Pointers to flexio bases for each instance.

const *clock_ip_name_t* s_flexioClocks[]

Pointers to flexio clocks for each instance.

void FLEXIO_SetPinConfig(FLEXIO_Type *base, uint32_t pin, *flexio_gpio_config_t* *config)

Configure a FLEXIO pin used by the board.

To Config the FLEXIO PIN, define a pin configuration, as either input or output, in the user file. Then, call the FLEXIO_SetPinConfig() function.

This is an example to define an input pin or an output pin configuration.

```
Define a digital input pin configuration,
flexio_gpio_config_t config =
{
  kFLEXIO_DigitalInput,
  0U,
  kFLEXIO_FlagRisingEdgeEnable | kFLEXIO_InputInterruptEnable,
}
Define a digital output pin configuration,
flexio_gpio_config_t config =
{
  kFLEXIO_DigitalOutput,
  0U,
  0U
}
```

Parameters

- base – FlexIO peripheral base address
- pin – FLEXIO pin number.
- config – FLEXIO pin configuration pointer.

FLEXIO_TIMER_TRIGGER_SEL_PININPUT(x)

Calculate FlexIO timer trigger.

FLEXIO_TIMER_TRIGGER_SEL_SHIFTnSTAT(x)

FLEXIO_TIMER_TRIGGER_SEL_TIMn(x)

struct _flexio_config_

#include <fsl_flexio.h> Define FlexIO user configuration structure.

Public Members

bool enableFlexio

Enable/disable FlexIO module

bool enableInDoze

Enable/disable FlexIO operation in doze mode

bool enableInDebug

Enable/disable FlexIO operation in debug mode

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

struct _flexio_timer_config

#include <fsl_flexio.h> Define FlexIO timer configuration structure.

Public Members

uint32_t triggerSelect
The internal trigger selection number using MACROs.

flexio_timer_trigger_polarity_t triggerPolarity
Trigger Polarity.

flexio_timer_trigger_source_t triggerSource
Trigger Source, internal (see 'trgsel') or external.

flexio_pin_config_t pinConfig
Timer Pin Configuration.

uint32_t pinSelect
Timer Pin number Select.

flexio_pin_polarity_t pinPolarity
Timer Pin Polarity.

flexio_timer_mode_t timerMode
Timer work Mode.

flexio_timer_output_t timerOutput
Configures the initial state of the Timer Output and whether it is affected by the Timer reset.

flexio_timer_decrement_source_t timerDecrement
Configures the source of the Timer decrement and the source of the Shift clock.

flexio_timer_reset_condition_t timerReset
Configures the condition that causes the timer counter (and optionally the timer output) to be reset.

flexio_timer_disable_condition_t timerDisable
Configures the condition that causes the Timer to be disabled and stop decrementing.

flexio_timer_enable_condition_t timerEnable
Configures the condition that causes the Timer to be enabled and start decrementing.

flexio_timer_stop_bit_condition_t timerStop
Timer STOP Bit generation.

flexio_timer_start_bit_condition_t timerStart
Timer STRAT Bit generation.

uint32_t timerCompare
Value for Timer Compare N Register.

struct __flexio__shifter__config
#include <fsl_flexio.h> Define FlexIO shifter configuration structure.

Public Members

uint32_t timerSelect
Selects which Timer is used for controlling the logic/shift register and generating the Shift clock.

flexio_shifter_timer_polarity_t timerPolarity
Timer Polarity.

flexio_pin_config_t pinConfig
Shifter Pin Configuration.

uint32_t pinSelect
Shifter Pin number Select.

flexio_pin_polarity_t pinPolarity
Shifter Pin Polarity.

flexio_shifter_mode_t shifterMode
Configures the mode of the Shifter.

uint32_t parallelWidth
Configures the parallel width when using parallel mode.

flexio_shifter_input_source_t inputSource
Selects the input source for the shifter.

flexio_shifter_stop_bit_t shifterStop
Shifter STOP bit.

flexio_shifter_start_bit_t shifterStart
Shifter START bit.

struct _flexio_gpio_config
#include <fsl_flexio.h> The FLEXIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, use inputConfig param. If configured as an output pin, use outputLogic.

Public Members

flexio_gpio_direction_t pinDirection
FLEXIO pin direction, input or output

uint8_t outputLogic
Set a default output logic, which has no use in input

uint8_t inputConfig
Set an input config

2.16 FlexIO eDMA SPI Driver

status_t FLEXIO_SPI_MasterTransferCreateHandleEDMA(*FLEXIO_SPI_Type* *base, *flexio_spi_master_edma_handle_t* *handle, *flexio_spi_master_edma_transfer_callback_t* callback, void *userData, *edma_handle_t* *txHandle, *edma_handle_t* *rxHandle)

Initializes the FlexIO SPI master eDMA handle.

This function initializes the FlexIO SPI master eDMA handle which can be used for other FlexIO SPI master transactional APIs. For a specified FlexIO SPI instance, call this API once to get the initialized handle.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – Pointer to flexio_spi_master_edma_handle_t structure to store the transfer state.
- callback – SPI callback, NULL means no callback.
- userData – callback function parameter.
- txHandle – User requested eDMA handle for FlexIO SPI RX eDMA transfer.
- rxHandle – User requested eDMA handle for FlexIO SPI TX eDMA transfer.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO SPI eDMA type/handle table out of range.

status_t FLEXIO_SPI_MasterTransferEDMA(*FLEXIO_SPI_Type* *base, *flexio_spi_master_edma_handle_t* *handle, *flexio_spi_transfer_t* *xfer)

Performs a non-blocking FlexIO SPI transfer using eDMA.

Note: This interface returns immediately after transfer initiates. Call FLEXIO_SPI_MasterGetTransferCountEDMA to poll the transfer status and check whether the FlexIO SPI transfer is finished.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – Pointer to flexio_spi_master_edma_handle_t structure to store the transfer state.
- xfer – Pointer to FlexIO SPI transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_FLEXIO_SPI_Busy – FlexIO SPI is not idle, is running another transfer.

void FLEXIO_SPI_MasterTransferAbortEDMA(*FLEXIO_SPI_Type* *base, *flexio_spi_master_edma_handle_t* *handle)

Aborts a FlexIO SPI transfer using eDMA.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – FlexIO SPI eDMA handle pointer.

status_t FLEXIO_SPI_MasterTransferGetCountEDMA(*FLEXIO_SPI_Type* *base, *flexio_spi_master_edma_handle_t* *handle, size_t *count)

Gets the number of bytes transferred so far using FlexIO SPI master eDMA.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – FlexIO SPI eDMA handle pointer.
- count – Number of bytes transferred so far by the non-blocking transaction.

static inline void FLEXIO_SPI_SlaveTransferCreateHandleEDMA(*FLEXIO_SPI_Type* *base, *flexio_spi_slave_edma_handle_t* *handle, *flexio_spi_slave_edma_transfer_callback_t* callback, void *userData, *edma_handle_t* *txHandle, *edma_handle_t* *rxHandle)

Initializes the FlexIO SPI slave eDMA handle.

This function initializes the FlexIO SPI slave eDMA handle.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – Pointer to flexio_spi_slave_edma_handle_t structure to store the transfer state.
- callback – SPI callback, NULL means no callback.
- userData – callback function parameter.
- txHandle – User requested eDMA handle for FlexIO SPI TX eDMA transfer.
- rxHandle – User requested eDMA handle for FlexIO SPI RX eDMA transfer.

status_t FLEXIO_SPI_SlaveTransferEDMA(*FLEXIO_SPI_Type* *base, *flexio_spi_slave_edma_handle_t* *handle, *flexio_spi_transfer_t* *xfer)

Performs a non-blocking FlexIO SPI transfer using eDMA.

Note: This interface returns immediately after transfer initiates. Call FLEXIO_SPI_SlaveGetTransferCountEDMA to poll the transfer status and check whether the FlexIO SPI transfer is finished.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – Pointer to flexio_spi_slave_edma_handle_t structure to store the transfer state.
- xfer – Pointer to FlexIO SPI transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_FLEXIO_SPI_Busy – FlexIO SPI is not idle, is running another transfer.

static inline void FLEXIO_SPI_SlaveTransferAbortEDMA(*FLEXIO_SPI_Type* *base, *flexio_spi_slave_edma_handle_t* *handle)

Aborts a FlexIO SPI transfer using eDMA.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – Pointer to flexio_spi_slave_edma_handle_t structure to store the transfer state.

static inline *status_t* FLEXIO_SPI_SlaveTransferGetCountEDMA(*FLEXIO_SPI_Type* *base, *flexio_spi_slave_edma_handle_t* *handle, size_t *count)

Gets the number of bytes transferred so far using FlexIO SPI slave eDMA.

Parameters

- base – Pointer to FLEXIO_SPI_Type structure.
- handle – FlexIO SPI eDMA handle pointer.
- count – Number of bytes transferred so far by the non-blocking transaction.

FSL_FLEXIO_SPI_EDMA_DRIVER_VERSION

FlexIO SPI EDMA driver version.

typedef struct *_flexio_spi_master_edma_handle* flexio_spi_master_edma_handle_t

typedef for flexio_spi_master_edma_handle_t in advance.

typedef *flexio_spi_master_edma_handle_t* flexio_spi_slave_edma_handle_t

Slave handle is the same with master handle.

typedef void (*flexio_spi_master_edma_transfer_callback_t)(*FLEXIO_SPI_Type* *base, *flexio_spi_master_edma_handle_t* *handle, *status_t* status, void *userData)

FlexIO SPI master callback for finished transmit.

typedef void (*flexio_spi_slave_edma_transfer_callback_t)(*FLEXIO_SPI_Type* *base, *flexio_spi_slave_edma_handle_t* *handle, *status_t* status, void *userData)

FlexIO SPI slave callback for finished transmit.

struct _flexio_spi_master_edma_handle

#include <fsl_flexio_spi_edma.h> FlexIO SPI eDMA transfer handle, users should not touch the content of the handle.

Public Members

size_t transferSize

Total bytes to be transferred.

uint8_t nbytes

eDMA minor byte transfer count initially configured.

bool txInProgress

Send transfer in progress

bool rxInProgress

Receive transfer in progress

edma_handle_t *txHandle

DMA handler for SPI send

edma_handle_t *rxHandle

DMA handler for SPI receive

flexio_spi_master_edma_transfer_callback_t callback

Callback for SPI DMA transfer

void *userData

User Data for SPI DMA callback

2.17 FlexIO eDMA UART Driver

status_t FLEXIO_UART_TransferCreateHandleEDMA(*FLEXIO_UART_Type* *base, *flexio_uart_edma_handle_t* *handle, *flexio_uart_edma_transfer_callback_t* callback, void *userData, *edma_handle_t* *txEdmaHandle, *edma_handle_t* *rxEdmaHandle)

Initializes the UART handle which is used in transactional functions.

Parameters

- base – Pointer to FLEXIO_UART_Type.
- handle – Pointer to flexio_uart_edma_handle_t structure.
- callback – The callback function.
- userData – The parameter of the callback function.
- rxEdmaHandle – User requested DMA handle for RX DMA transfer.
- txEdmaHandle – User requested DMA handle for TX DMA transfer.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO SPI eDMA type/handle table out of range.

status_t FLEXIO_UART_TransferSendEDMA(*FLEXIO_UART_Type* *base, *flexio_uart_edma_handle_t* *handle, *flexio_uart_transfer_t* *xfer)

Sends data using eDMA.

This function sends data using eDMA. This is a non-blocking function, which returns right away. When all data is sent out, the send callback function is called.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – UART handle pointer.
- xfer – UART eDMA transfer structure, see flexio_uart_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_FLEXIO_UART_TxBusy – Previous transfer on going.

status_t FLEXIO_UART_TransferReceiveEDMA(*FLEXIO_UART_Type* *base, *flexio_uart_edma_handle_t* *handle, *flexio_uart_transfer_t* *xfer)

Receives data using eDMA.

This function receives data using eDMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – Pointer to flexio_uart_edma_handle_t structure
- xfer – UART eDMA transfer structure, see flexio_uart_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_UART_RxBusy – Previous transfer on going.

void FLEXIO_UART_TransferAbortSendEDMA(*FLEXIO_UART_Type* *base, *flexio_uart_edma_handle_t* *handle)

Aborts the sent data which using eDMA.

This function aborts sent data which using eDMA.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – Pointer to flexio_uart_edma_handle_t structure

void FLEXIO_UART_TransferAbortReceiveEDMA(*FLEXIO_UART_Type* *base, *flexio_uart_edma_handle_t* *handle)

Aborts the receive data which using eDMA.

This function aborts the receive data which using eDMA.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – Pointer to flexio_uart_edma_handle_t structure

status_t FLEXIO_UART_TransferGetSendCountEDMA(*FLEXIO_UART_Type* *base, *flexio_uart_edma_handle_t* *handle, size_t *count)

Gets the number of bytes sent out.

This function gets the number of bytes sent out.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – Pointer to flexio_uart_edma_handle_t structure
- count – Number of bytes sent so far by the non-blocking transaction.

Return values

- kStatus_NoTransferInProgress – transfer has finished or no transfer in progress.
- kStatus_Success – Successfully return the count.

status_t FLEXIO_UART_TransferGetReceiveCountEDMA(*FLEXIO_UART_Type* *base, *flexio_uart_edma_handle_t* *handle, size_t *count)

Gets the number of bytes received.

This function gets the number of bytes received.

Parameters

- base – Pointer to FLEXIO_UART_Type
- handle – Pointer to flexio_uart_edma_handle_t structure
- count – Number of bytes received so far by the non-blocking transaction.

Return values

- kStatus_NoTransferInProgress – transfer has finished or no transfer in progress.
- kStatus_Success – Successfully return the count.

FSL_FLEXIO_UART_EDMA_DRIVER_VERSION
FlexIO UART EDMA driver version.

typedef struct *_flexio_uart_edma_handle* flexio_uart_edma_handle_t

typedef void (*flexio_uart_edma_transfer_callback_t)(*FLEXIO_UART_Type* *base, *flexio_uart_edma_handle_t* *handle, *status_t* status, void *userData)
UART transfer callback function.

struct _flexio_uart_edma_handle
#include <fsl_flexio_uart_edma.h> UART eDMA handle.

Public Members

flexio_uart_edma_transfer_callback_t callback
Callback function.

void *userData
UART callback function parameter.

size_t txDataSizeAll
Total bytes to be sent.

size_t rxDataSizeAll
Total bytes to be received.

edma_handle_t *txEdmaHandle
The eDMA TX channel used.

edma_handle_t *rxEdmaHandle
The eDMA RX channel used.

uint8_t nbytes
eDMA minor byte transfer count initially configured.

volatile uint8_t txState
TX transfer state.

volatile uint8_t rxState
RX transfer state

2.18 FlexIO I2C Master Driver

status_t FLEXIO_I2C_CheckForBusyBus(*FLEXIO_I2C_Type* *base)
Make sure the bus isn't already pulled down.

Check the FLEXIO pin status to see whether either of SDA and SCL pin is pulled down.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure..

Return values

- kStatus_Success –
- kStatus_FLEXIO_I2C_Busy –

status_t FLEXIO_I2C_MasterInit(*FLEXIO_I2C_Type* *base, *flexio_i2c_master_config_t* *masterConfig, uint32_t srcClock_Hz)

Ungates the FlexIO clock, resets the FlexIO module, and configures the FlexIO I2C hardware configuration.

Example

```
FLEXIO_I2C_Type base = {
.flexioBase = FLEXIO,
.SDAPinIndex = 0,
.SCLPinIndex = 1,
.shifterIndex = {0,1},
.timerIndex = {0,1}
};
flexio_i2c_master_config_t config = {
.enableInDoze = false,
.enableInDebug = true,
.enableFastAccess = false,
.baudRate_Bps = 100000
};
FLEXIO_I2C_MasterInit(base, &config, srcClock_Hz);
```

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- masterConfig – Pointer to flexio_i2c_master_config_t structure.
- srcClock_Hz – FlexIO source clock in Hz.

Return values

- kStatus_Success – Initialization successful
- kStatus_InvalidArgument – The source clock exceed upper range limitation

void FLEXIO_I2C_MasterDeinit(*FLEXIO_I2C_Type* *base)

De-initializes the FlexIO I2C master peripheral. Calling this API Resets the FlexIO I2C master shifer and timer config, module can't work unless the FLEXIO_I2C_MasterInit is called.

Parameters

- base – pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterGetDefaultConfig(*flexio_i2c_master_config_t* *masterConfig)

Gets the default configuration to configure the FlexIO module. The configuration can be used directly for calling the FLEXIO_I2C_MasterInit().

Example:

```
flexio_i2c_master_config_t config;
FLEXIO_I2C_MasterGetDefaultConfig(&config);
```

Parameters

- masterConfig – Pointer to flexio_i2c_master_config_t structure.

static inline void FLEXIO_I2C_MasterEnable(*FLEXIO_I2C_Type* *base, bool enable)

Enables/disables the FlexIO module operation.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- enable – Pass true to enable module, false does not have any effect.

uint32_t FLEXIO_I2C_MasterGetStatusFlags(*FLEXIO_I2C_Type* *base)

Gets the FlexIO I2C master status flags.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure

Returns

Status flag, use status flag to AND _flexio_i2c_master_status_flags can get the related status.

void FLEXIO_I2C_MasterClearStatusFlags(*FLEXIO_I2C_Type* *base, uint32_t mask)

Clears the FlexIO I2C master status flags.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- mask – Status flag. The parameter can be any combination of the following values:
  - kFLEXIO_I2C_RxFullFlag
  - kFLEXIO_I2C_ReceiveNakFlag

void FLEXIO_I2C_MasterEnableInterrupts(*FLEXIO_I2C_Type* *base, uint32_t mask)

Enables the FlexIO i2c master interrupt requests.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- mask – Interrupt source. Currently only one interrupt request source:
  - kFLEXIO_I2C_TransferCompleteInterruptEnable

void FLEXIO_I2C_MasterDisableInterrupts(*FLEXIO_I2C_Type* *base, uint32_t mask)

Disables the FlexIO I2C master interrupt requests.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- mask – Interrupt source.

void FLEXIO_I2C_MasterSetBaudRate(*FLEXIO_I2C_Type* *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)

Sets the FlexIO I2C master transfer baudrate.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure
- baudRate_Bps – the baud rate value in HZ
- srcClock_Hz – source clock in HZ

void FLEXIO_I2C_MasterStart(*FLEXIO_I2C_Type* *base, uint8_t address, *flexio_i2c_direction_t* direction)

Sends START + 7-bit address to the bus.

Note: This API should be called when the transfer configuration is ready to send a START signal and 7-bit address to the bus. This is a non-blocking API, which returns directly after the address is put into the data register but the address transfer is not finished on the bus. Ensure that the kFLEXIO_I2C_RxFullFlag status is asserted before calling this API.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- address – 7-bit address.
- direction – transfer direction. This parameter is one of the values in flexio_i2c_direction_t:
  - kFLEXIO_I2C_Write: Transmit
  - kFLEXIO_I2C_Read: Receive

void FLEXIO_I2C_MasterStop(*FLEXIO_I2C_Type* *base)

Sends the stop signal on the bus.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterRepeatedStart(*FLEXIO_I2C_Type* *base)

Sends the repeated start signal on the bus.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterAbortStop(*FLEXIO_I2C_Type* *base)

Sends the stop signal when transfer is still on-going.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

void FLEXIO_I2C_MasterEnableAck(*FLEXIO_I2C_Type* *base, bool enable)

Configures the sent ACK/NAK for the following byte.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- enable – True to configure send ACK, false configure to send NAK.

status_t FLEXIO_I2C_MasterSetTransferCount(*FLEXIO_I2C_Type* *base, uint16_t count)

Sets the number of bytes to be transferred from a start signal to a stop signal.

Note: Call this API before a transfer begins because the timer generates a number of clocks according to the number of bytes that need to be transferred.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- count – Number of bytes need to be transferred from a start signal to a re-start/stop signal

Return values

- kStatus_Success – Successfully configured the count.
- kStatus_InvalidArgument – Input argument is invalid.

static inline void FLEXIO_I2C_MasterWriteByte(*FLEXIO_I2C_Type* *base, uint32_t data)

Writes one byte of data to the I2C bus.

Note: This is a non-blocking API, which returns directly after the data is put into the data register but the data transfer is not finished on the bus. Ensure that the TxEmptyFlag is asserted before calling this API.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- data – a byte of data.

static inline uint8_t FLEXIO_I2C_MasterReadByte(*FLEXIO_I2C_Type* *base)

Reads one byte of data from the I2C bus.

Note: This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the data is ready in the register.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.

Returns

data byte read.

status_t FLEXIO_I2C_MasterWriteBlocking(*FLEXIO_I2C_Type* *base, const uint8_t *txBuff, uint8_t txSize)

Sends a buffer of data in bytes.

Note: This function blocks via polling until all bytes have been sent.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- txBuff – The data bytes to send.
- txSize – The number of data bytes to send.

Return values

- kStatus_Success – Successfully write data.
- kStatus_FLEXIO_I2C_Nak – Receive NAK during writing data.
- kStatus_FLEXIO_I2C_Timeout – Timeout polling status flags.

status_t FLEXIO_I2C_MasterReadBlocking(*FLEXIO_I2C_Type* *base, uint8_t *rxBuff, uint8_t rxSize)

Receives a buffer of bytes.

Note: This function blocks via polling until all bytes have been received.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- rxBuff – The buffer to store the received bytes.
- rxSize – The number of data bytes to be received.

Return values

- kStatus_Success – Successfully read data.
- kStatus_FLEXIO_I2C_Timeout – Timeout polling status flags.

status_t FLEXIO_I2C_MasterTransferBlocking(*FLEXIO_I2C_Type* *base, *flexio_i2c_master_transfer_t* *xfer)

Performs a master polling transfer on the I2C bus.

Note: The API does not return until the transfer succeeds or fails due to receiving NAK.

Parameters

- base – pointer to FLEXIO_I2C_Type structure.
- xfer – pointer to flexio_i2c_master_transfer_t structure.

Returns

status of status_t.

status_t FLEXIO_I2C_MasterTransferCreateHandle(*FLEXIO_I2C_Type* *base, *flexio_i2c_master_handle_t* *handle, *flexio_i2c_master_transfer_callback_t* callback, void *userData)

Initializes the I2C handle which is used in transactional functions.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- handle – Pointer to flexio_i2c_master_handle_t structure to store the transfer state.
- callback – Pointer to user callback function.
- userData – User param passed to the callback function.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/isr table out of range.

status_t FLEXIO_I2C_MasterTransferNonBlocking(*FLEXIO_I2C_Type* *base, *flexio_i2c_master_handle_t* *handle, *flexio_i2c_master_transfer_t* *xfer)

Performs a master interrupt non-blocking transfer on the I2C bus.

Note: The API returns immediately after the transfer initiates. Call FLEXIO_I2C_MasterTransferGetCount to poll the transfer status to check whether the transfer is finished. If the return status is not kStatus_FLEXIO_I2C_Busy, the transfer is finished.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure
- handle – Pointer to flexio_i2c_master_handle_t structure which stores the transfer state
- xfer – pointer to flexio_i2c_master_transfer_t structure

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_FLEXIO_I2C_Busy – FlexIO I2C is not idle, is running another transfer.

status_t FLEXIO_I2C_MasterTransferGetCount(*FLEXIO_I2C_Type* *base, *flexio_i2c_master_handle_t* *handle, size_t *count)

Gets the master transfer status during a interrupt non-blocking transfer.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure.
- handle – Pointer to flexio_i2c_master_handle_t structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_InvalidArgument – count is Invalid.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.
- kStatus_Success – Successfully return the count.

void FLEXIO_I2C_MasterTransferAbort(*FLEXIO_I2C_Type* *base, *flexio_i2c_master_handle_t* *handle)

Aborts an interrupt non-blocking transfer early.

Note: This API can be called at any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- base – Pointer to FLEXIO_I2C_Type structure
- handle – Pointer to flexio_i2c_master_handle_t structure which stores the transfer state

void FLEXIO_I2C_MasterTransferHandleIRQ(void *i2cType, void *i2cHandle)

Master interrupt handler.

Parameters

- i2cType – Pointer to FLEXIO_I2C_Type structure
- i2cHandle – Pointer to flexio_i2c_master_transfer_t structure

FSL_FLEXIO_I2C_MASTER_DRIVER_VERSION

FlexIO I2C transfer status.

Values:

enumerator kStatus_FLEXIO_I2C_Busy
I2C is busy doing transfer.

enumerator kStatus_FLEXIO_I2C_Idle
I2C is busy doing transfer.

enumerator kStatus_FLEXIO_I2C_Nak
NAK received during transfer.

enumerator kStatus_FLEXIO_I2C_Timeout
Timeout polling status flags.

enum _flexio_i2c_master_interrupt

Define FlexIO I2C master interrupt mask.

Values:

enumerator kFLEXIO_I2C_TxEmptyInterruptEnable

Tx buffer empty interrupt enable.

enumerator kFLEXIO_I2C_RxFullInterruptEnable

Rx buffer full interrupt enable.

enum _flexio_i2c_master_status_flags

Define FlexIO I2C master status mask.

Values:

enumerator kFLEXIO_I2C_TxEmptyFlag

Tx shifter empty flag.

enumerator kFLEXIO_I2C_RxFullFlag

Rx shifter full/Transfer complete flag.

enumerator kFLEXIO_I2C_ReceiveNakFlag

Receive NAK flag.

enum _flexio_i2c_direction

Direction of master transfer.

Values:

enumerator kFLEXIO_I2C_Write

Master send to slave.

enumerator kFLEXIO_I2C_Read

Master receive from slave.

typedef enum *_flexio_i2c_direction* flexio_i2c_direction_t

Direction of master transfer.

typedef struct *_flexio_i2c_type* FLEXIO_I2C_Type

Define FlexIO I2C master access structure typedef.

typedef struct *_flexio_i2c_master_config* flexio_i2c_master_config_t

Define FlexIO I2C master user configuration structure.

typedef struct *_flexio_i2c_master_transfer* flexio_i2c_master_transfer_t

Define FlexIO I2C master transfer structure.

typedef struct *_flexio_i2c_master_handle* flexio_i2c_master_handle_t

FlexIO I2C master handle typedef.

typedef void (*flexio_i2c_master_transfer_callback_t)(*FLEXIO_I2C_Type* *base, *flexio_i2c_master_handle_t* *handle, *status_t* status, void *userData)

FlexIO I2C master transfer callback typedef.

I2C_RETRY_TIMES

Retry times for waiting flag.

struct _flexio_i2c_type

#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master access structure typedef.

Public Members

FLEXIO_Type *flexioBase
: FlexIO base pointer.

uint8_t SDAPinIndex
: Pin select for I2C SDA.

uint8_t SCLPinIndex
: Pin select for I2C SCL.

uint8_t shifterIndex[2]
: Shifter index used in FlexIO I2C.

uint8_t timerIndex[3]
: Timer index used in FlexIO I2C.

uint32_t baudrate
: Master transfer baudrate, used to calculate delay time.

struct __flexio_i2c_master_config
: *#include <fsl_flexio_i2c_master.h>* Define FlexIO I2C master user configuration structure.

Public Members

bool enableMaster
: Enables the FlexIO I2C peripheral at initialization time.

bool enableInDoze
: Enable/disable FlexIO operation in doze mode.

bool enableInDebug
: Enable/disable FlexIO operation in debug mode.

bool enableFastAccess
: Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

uint32_t baudRate_Bps
: Baud rate in Bps.

struct __flexio_i2c_master_transfer
: *#include <fsl_flexio_i2c_master.h>* Define FlexIO I2C master transfer structure.

Public Members

uint32_t flags
: Transfer flag which controls the transfer, reserved for FlexIO I2C.

uint8_t slaveAddress
: 7-bit slave address.

flexio_i2c_direction_t direction
: Transfer direction, read or write.

uint32_t subaddress
: Sub address. Transferred MSB first.

uint8_t subaddressSize
: Size of sub address.

uint8_t volatile *data
Transfer buffer.

volatile size_t dataSize
Transfer size.

struct __flexio_i2c_master_handle
#include <fsl_flexio_i2c_master.h> Define FlexIO I2C master handle structure.

Public Members

flexio_i2c_master_transfer_t transfer
FlexIO I2C master transfer copy.

size_t transferSize
Total bytes to be transferred.

uint8_t state
Transfer state maintained during transfer.

flexio_i2c_master_transfer_callback_t completionCallback
Callback function called at transfer event. Callback function called at transfer event.

void *userData
Callback parameter passed to callback function.

bool needRestart
Whether master needs to send re-start signal.

2.19 FlexIO I2S Driver

void FLEXIO_I2S_Init(*FLEXIO_I2S_Type* *base, const *flexio_i2s_config_t* *config)
Initializes the FlexIO I2S.

This API configures FlexIO pins and shifter to I2S and configures the FlexIO I2S with a configuration structure. The configuration structure can be filled by the user, or be set with default values by FLEXIO_I2S_GetDefaultConfig().

Note: This API should be called at the beginning of the application to use the FlexIO I2S driver. Otherwise, any access to the FlexIO I2S module can cause hard fault because the clock is not enabled.

Parameters

- base – FlexIO I2S base pointer
- config – FlexIO I2S configure structure.

void FLEXIO_I2S_GetDefaultConfig(*flexio_i2s_config_t* *config)
Sets the FlexIO I2S configuration structure to default values.

The purpose of this API is to get the configuration structure initialized for use in FLEXIO_I2S_Init(). Users may use the initialized structure unchanged in FLEXIO_I2S_Init() or modify some fields of the structure before calling FLEXIO_I2S_Init().

Parameters

- config – pointer to master configuration structure

void FLEXIO_I2S_Deinit(*FLEXIO_I2S_Type* *base)

De-initializes the FlexIO I2S.

Calling this API resets the FlexIO I2S shifter and timer config. After calling this API, call the FLEXO_I2S_Init to use the FlexIO I2S module.

Parameters

- base – FlexIO I2S base pointer

static inline void FLEXIO_I2S_Enable(*FLEXIO_I2S_Type* *base, bool enable)

Enables/disables the FlexIO I2S module operation.

Parameters

- base – Pointer to FLEXIO_I2S_Type
- enable – True to enable, false dose not have any effect.

uint32_t FLEXIO_I2S_GetStatusFlags(*FLEXIO_I2S_Type* *base)

Gets the FlexIO I2S status flags.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure

Returns

Status flag, which are ORed by the enumerators in the _flexio_i2s_status_flags.

void FLEXIO_I2S_EnableInterrupts(*FLEXIO_I2S_Type* *base, uint32_t mask)

Enables the FlexIO I2S interrupt.

This function enables the FlexIO UART interrupt.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure
- mask – interrupt source

void FLEXIO_I2S_DisableInterrupts(*FLEXIO_I2S_Type* *base, uint32_t mask)

Disables the FlexIO I2S interrupt.

This function enables the FlexIO UART interrupt.

Parameters

- base – pointer to FLEXIO_I2S_Type structure
- mask – interrupt source

static inline void FLEXIO_I2S_TxEnableDMA(*FLEXIO_I2S_Type* *base, bool enable)

Enables/disables the FlexIO I2S Tx DMA requests.

Parameters

- base – FlexIO I2S base pointer
- enable – True means enable DMA, false means disable DMA.

static inline void FLEXIO_I2S_RxEnableDMA(*FLEXIO_I2S_Type* *base, bool enable)

Enables/disables the FlexIO I2S Rx DMA requests.

Parameters

- base – FlexIO I2S base pointer
- enable – True means enable DMA, false means disable DMA.

static inline uint32_t FLEXIO_I2S_TxGetDataRegisterAddress(*FLEXIO_I2S_Type* *base)

Gets the FlexIO I2S send data register address.

This function returns the I2S data register address, mainly used by DMA/eDMA.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure

Returns

FlexIO i2s send data register address.

static inline uint32_t FLEXIO_I2S_RxGetDataRegisterAddress(*FLEXIO_I2S_Type* *base)

Gets the FlexIO I2S receive data register address.

This function returns the I2S data register address, mainly used by DMA/eDMA.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure

Returns

FlexIO i2s receive data register address.

void FLEXIO_I2S_MasterSetFormat(*FLEXIO_I2S_Type* *base, *flexio_i2s_format_t* *format, uint32_t srcClock_Hz)

Configures the FlexIO I2S audio format in master mode.

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure
- format – Pointer to FlexIO I2S audio data format structure.
- srcClock_Hz – I2S master clock source frequency in Hz.

void FLEXIO_I2S_SlaveSetFormat(*FLEXIO_I2S_Type* *base, *flexio_i2s_format_t* *format)

Configures the FlexIO I2S audio format in slave mode.

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure
- format – Pointer to FlexIO I2S audio data format structure.

status_t FLEXIO_I2S_WriteBlocking(*FLEXIO_I2S_Type* *base, uint8_t bitWidth, uint8_t *txData, size_t size)

Sends data using a blocking method.

Note: This function blocks via polling until data is ready to be sent.

Parameters

- base – FlexIO I2S base pointer.
- bitWidth – How many bits in a audio word, usually 8/16/24/32 bits.
- txData – Pointer to the data to be written.
- size – Bytes to be written.

Return values

- kStatus_Success – Successfully write data.
- kStatus_FLEXIO_I2C_Timeout – Timeout polling status flags.

static inline void FLEXIO_I2S_WriteData(*FLEXIO_I2S_Type* *base, uint8_t bitWidth, uint32_t data)

Writes data into a data register.

Parameters

- base – FlexIO I2S base pointer.
- bitWidth – How many bits in a audio word, usually 8/16/24/32 bits.
- data – Data to be written.

status_t FLEXIO_I2S_ReadBlocking(*FLEXIO_I2S_Type* *base, uint8_t bitWidth, uint8_t *rxData, size_t size)

Receives a piece of data using a blocking method.

Note: This function blocks via polling until data is ready to be sent.

Parameters

- base – FlexIO I2S base pointer
- bitWidth – How many bits in a audio word, usually 8/16/24/32 bits.
- rxData – Pointer to the data to be read.
- size – Bytes to be read.

Return values

- kStatus_Success – Successfully read data.
- kStatus_FLEXIO_I2C_Timeout – Timeout polling status flags.

static inline uint32_t FLEXIO_I2S_ReadData(*FLEXIO_I2S_Type* *base)

Reads a data from the data register.

Parameters

- base – FlexIO I2S base pointer

Returns

Data read from data register.

void FLEXIO_I2S_TransferTxCreateHandle(*FLEXIO_I2S_Type* *base, *flexio_i2s_handle_t* *handle, *flexio_i2s_callback_t* callback, void *userData)

Initializes the FlexIO I2S handle.

This function initializes the FlexIO I2S handle which can be used for other FlexIO I2S transactional APIs. Call this API once to get the initialized handle.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure
- handle – Pointer to flexio_i2s_handle_t structure to store the transfer state.
- callback – FlexIO I2S callback function, which is called while finished a block.
- userData – User parameter for the FlexIO I2S callback.

void FLEXIO_I2S_TransferSetFormat(*FLEXIO_I2S_Type* *base, *flexio_i2s_handle_t* *handle, *flexio_i2s_format_t* *format, uint32_t srcClock_Hz)

Configures the FlexIO I2S audio format.

Audio format can be changed at run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure.
- handle – FlexIO I2S handle pointer.
- format – Pointer to audio data format structure.
- srcClock_Hz – FlexIO I2S bit clock source frequency in Hz. This parameter should be 0 while in slave mode.

void FLEXIO_I2S_TransferRxCreateHandle(*FLEXIO_I2S_Type* *base, *flexio_i2s_handle_t* *handle, *flexio_i2s_callback_t* callback, void *userData)

Initializes the FlexIO I2S receive handle.

This function initializes the FlexIO I2S handle which can be used for other FlexIO I2S transactional APIs. Call this API once to get the initialized handle.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure.
- handle – Pointer to flexio_i2s_handle_t structure to store the transfer state.
- callback – FlexIO I2S callback function, which is called while finished a block.
- userData – User parameter for the FlexIO I2S callback.

status_t FLEXIO_I2S_TransferSendNonBlocking(*FLEXIO_I2S_Type* *base, *flexio_i2s_handle_t* *handle, *flexio_i2s_transfer_t* *xfer)

Performs an interrupt non-blocking send transfer on FlexIO I2S.

Note: The API returns immediately after transfer initiates. Call FLEXIO_I2S_GetRemainingBytes to poll the transfer status and check whether the transfer is finished. If the return status is 0, the transfer is finished.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure.
- handle – Pointer to flexio_i2s_handle_t structure which stores the transfer state
- xfer – Pointer to flexio_i2s_transfer_t structure

Return values

- kStatus_Success – Successfully start the data transmission.
- kStatus_FLEXIO_I2S_TxBusy – Previous transmission still not finished, data not all written to TX register yet.
- kStatus_InvalidArgument – The input parameter is invalid.

status_t FLEXIO_I2S_TransferReceiveNonBlocking(*FLEXIO_I2S_Type* *base, *flexio_i2s_handle_t* *handle, *flexio_i2s_transfer_t* *xfer)

Performs an interrupt non-blocking receive transfer on FlexIO I2S.

Note: The API returns immediately after transfer initiates. Call FLEXIO_I2S_GetRemainingBytes to poll the transfer status to check whether the transfer is finished. If the return status is 0, the transfer is finished.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure.
- handle – Pointer to flexio_i2s_handle_t structure which stores the transfer state
- xfer – Pointer to flexio_i2s_transfer_t structure

Return values

- kStatus_Success – Successfully start the data receive.
- kStatus_FLEXIO_I2S_RxBusy – Previous receive still not finished.
- kStatus_InvalidArgument – The input parameter is invalid.

void FLEXIO_I2S_TransferAbortSend(*FLEXIO_I2S_Type* *base, *flexio_i2s_handle_t* *handle)

Aborts the current send.

Note: This API can be called at any time when interrupt non-blocking transfer initiates to abort the transfer in a early time.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure.
- handle – Pointer to flexio_i2s_handle_t structure which stores the transfer state

void FLEXIO_I2S_TransferAbortReceive(*FLEXIO_I2S_Type* *base, *flexio_i2s_handle_t* *handle)

Aborts the current receive.

Note: This API can be called at any time when interrupt non-blocking transfer initiates to abort the transfer in a early time.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure.
- handle – Pointer to flexio_i2s_handle_t structure which stores the transfer state

status_t FLEXIO_I2S_TransferGetSendCount(*FLEXIO_I2S_Type* *base, *flexio_i2s_handle_t* *handle, size_t *count)

Gets the remaining bytes to be sent.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure.
- handle – Pointer to flexio_i2s_handle_t structure which stores the transfer state
- count – Bytes sent.

Return values

- kStatus_Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

status_t FLEXIO_I2S_TransferGetReceiveCount(*FLEXIO_I2S_Type* *base, *flexio_i2s_handle_t* *handle, size_t *count)

Gets the remaining bytes to be received.

Parameters

- base – Pointer to FLEXIO_I2S_Type structure.
- handle – Pointer to flexio_i2s_handle_t structure which stores the transfer state
- count – Bytes recieved.

Return values

- kStatus_Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

Returns

count Bytes received.

void FLEXIO_I2S_TransferTxHandleIRQ(void *i2sBase, void *i2sHandle)

Tx interrupt handler.

Parameters

- i2sBase – Pointer to FLEXIO_I2S_Type structure.
- i2sHandle – Pointer to flexio_i2s_handle_t structure

void FLEXIO_I2S_TransferRxHandleIRQ(void *i2sBase, void *i2sHandle)

Rx interrupt handler.

Parameters

- i2sBase – Pointer to FLEXIO_I2S_Type structure.
- i2sHandle – Pointer to flexio_i2s_handle_t structure.

FSL_FLEXIO_I2S_DRIVER_VERSION

FlexIO I2S driver version 2.2.2.

FlexIO I2S transfer status.

Values:

enumerator kStatus_FLEXIO_I2S_Idle

FlexIO I2S is in idle state

enumerator kStatus_FLEXIO_I2S_TxBusy

FlexIO I2S Tx is busy

enumerator kStatus_FLEXIO_I2S_RxBusy

FlexIO I2S Tx is busy

enumerator kStatus_FLEXIO_I2S_Error

FlexIO I2S error occurred

enumerator kStatus_FLEXIO_I2S_QueueFull

FlexIO I2S transfer queue is full.

enumerator kStatus_FLEXIO_I2S_Timeout
FlexIO I2S timeout polling status flags.

enum _flexio_i2s_master_slave
Master or slave mode.

Values:

enumerator kFLEXIO_I2S_Master
Master mode

enumerator kFLEXIO_I2S_Slave
Slave mode

_flexio_i2s_interrupt_enable Define FlexIO FlexIO I2S interrupt mask.

Values:

enumerator kFLEXIO_I2S_TxDataRegEmptyInterruptEnable
Transmit buffer empty interrupt enable.

enumerator kFLEXIO_I2S_RxDataRegFullInterruptEnable
Receive buffer full interrupt enable.

_flexio_i2s_status_flags Define FlexIO FlexIO I2S status mask.

Values:

enumerator kFLEXIO_I2S_TxDataRegEmptyFlag
Transmit buffer empty flag.

enumerator kFLEXIO_I2S_RxDataRegFullFlag
Receive buffer full flag.

enum _flexio_i2s_sample_rate
Audio sample rate.

Values:

enumerator kFLEXIO_I2S_SampleRate8KHz
Sample rate 8000Hz

enumerator kFLEXIO_I2S_SampleRate11025Hz
Sample rate 11025Hz

enumerator kFLEXIO_I2S_SampleRate12KHz
Sample rate 12000Hz

enumerator kFLEXIO_I2S_SampleRate16KHz
Sample rate 16000Hz

enumerator kFLEXIO_I2S_SampleRate22050Hz
Sample rate 22050Hz

enumerator kFLEXIO_I2S_SampleRate24KHz
Sample rate 24000Hz

enumerator kFLEXIO_I2S_SampleRate32KHz
Sample rate 32000Hz

enumerator kFLEXIO_I2S_SampleRate44100Hz
Sample rate 44100Hz

enumerator kFLEXIO_I2S_SampleRate48KHz
Sample rate 48000Hz

enumerator kFLEXIO_I2S_SampleRate96KHz
Sample rate 96000Hz

enum _flexio_i2s_word_width
Audio word width.

Values:

enumerator kFLEXIO_I2S_WordWidth8bits
Audio data width 8 bits

enumerator kFLEXIO_I2S_WordWidth16bits
Audio data width 16 bits

enumerator kFLEXIO_I2S_WordWidth24bits
Audio data width 24 bits

enumerator kFLEXIO_I2S_WordWidth32bits
Audio data width 32 bits

typedef struct *_flexio_i2s_type* FLEXIO_I2S_Type
Define FlexIO I2S access structure typedef.

typedef enum *_flexio_i2s_master_slave* flexio_i2s_master_slave_t
Master or slave mode.

typedef struct *_flexio_i2s_config* flexio_i2s_config_t
FlexIO I2S configure structure.

typedef struct *_flexio_i2s_format* flexio_i2s_format_t
FlexIO I2S audio format, FlexIO I2S only support the same format in Tx and Rx.

typedef enum *_flexio_i2s_sample_rate* flexio_i2s_sample_rate_t
Audio sample rate.

typedef enum *_flexio_i2s_word_width* flexio_i2s_word_width_t
Audio word width.

typedef struct *_flexio_i2s_transfer* flexio_i2s_transfer_t
Define FlexIO I2S transfer structure.

typedef struct *_flexio_i2s_handle* flexio_i2s_handle_t

typedef void (*flexio_i2s_callback_t)(*FLEXIO_I2S_Type* *base, *flexio_i2s_handle_t* *handle, *status_t* status, void *userData)
FlexIO I2S xfer callback prototype.

I2S_RETRY_TIMES
Retry times for waiting flag.

FLEXIO_I2S_XFER_QUEUE_SIZE
FlexIO I2S transfer queue size, user can refine it according to use case.

struct _flexio_i2s_type
#include <fsl_flexio_i2s.h> Define FlexIO I2S access structure typedef.

Public Members

FLEXIO_Type *flexioBase
FlexIO base pointer

uint8_t txPinIndex
Tx data pin index in FlexIO pins

uint8_t rxPinIndex
Rx data pin index

uint8_t bclkPinIndex
Bit clock pin index

uint8_t fsPinIndex
Frame sync pin index

uint8_t txShifterIndex
Tx data shifter index

uint8_t rxShifterIndex
Rx data shifter index

uint8_t bclkTimerIndex
Bit clock timer index

uint8_t fsTimerIndex
Frame sync timer index

struct __flexio_i2s_config
#include <fsl_flexio_i2s.h> FlexIO I2S configure structure.

Public Members

bool enableI2S
Enable FlexIO I2S

flexio_i2s_master_slave_t masterSlave
Master or slave

flexio_pin_polarity_t txPinPolarity
Tx data pin polarity, active high or low

flexio_pin_polarity_t rxPinPolarity
Rx data pin polarity

flexio_pin_polarity_t bclkPinPolarity
Bit clock pin polarity

flexio_pin_polarity_t fsPinPolarity
Frame sync pin polarity

flexio_shifter_timer_polarity_t txTimerPolarity
Tx data valid on bclk rising or falling edge

flexio_shifter_timer_polarity_t rxTimerPolarity
Rx data valid on bclk rising or falling edge

struct __flexio_i2s_format
#include <fsl_flexio_i2s.h> FlexIO I2S audio format, FlexIO I2S only support the same format in Tx and Rx.

Public Members

uint8_t bitWidth
Bit width of audio data, always 8/16/24/32 bits

uint32_t sampleRate_Hz
Sample rate of the audio data

struct _flexio_i2s_transfer
#include <fsl_flexio_i2s.h> Define FlexIO I2S transfer structure.

Public Members

uint8_t *data
Data buffer start pointer

size_t dataSize
Bytes to be transferred.

struct _flexio_i2s_handle
#include <fsl_flexio_i2s.h> Define FlexIO I2S handle structure.

Public Members

uint32_t state
Internal state

flexio_i2s_callback_t callback
Callback function called at transfer event

void *userData
Callback parameter passed to callback function

uint8_t bitWidth
Bit width for transfer, 8/16/24/32bits

flexio_i2s_transfer_t queue[(4U)]
Transfer queue storing queued transfer

size_t transferSize[(4U)]
Data bytes need to transfer

volatile uint8_t queueUser
Index for user to queue transfer

volatile uint8_t queueDriver
Index for driver to get the transfer data and size

2.20 FlexIO SPI Driver

void FLEXIO_SPI_MasterInit(*FLEXIO_SPI_Type* *base, *flexio_spi_master_config_t* *masterConfig, uint32_t srcClock_Hz)

Ungates the FlexIO clock, resets the FlexIO module, configures the FlexIO SPI master hardware, and configures the FlexIO SPI with FlexIO SPI master configuration. The configuration structure can be filled by the user, or be set with default values by the FLEXIO_SPI_MasterGetDefaultConfig().

Example

```
FLEXIO_SPI_Type spiDev = {
.flexioBase = FLEXIO,
.SDOPinIndex = 0,
.SDIPinIndex = 1,
.SCKPinIndex = 2,
.CSnPinIndex = 3,
.shifterIndex = {0,1},
.timerIndex = {0,1}
};
flexio_spi_master_config_t config = {
.enableMaster = true,
.enableInDoze = false,
.enableInDebug = true,
.enableFastAccess = false,
.baudRate_Bps = 500000,
.phase = kFLEXIO_SPI_ClockPhaseFirstEdge,
.direction = kFLEXIO_SPI_MsbFirst,
.dataMode = kFLEXIO_SPI_8BitMode
};
FLEXIO_SPI_MasterInit(&spiDev, &config, srcClock_Hz);
```

Note: 1.FlexIO SPI master only support CPOL = 0, which means clock inactive low. 2.For FlexIO SPI master, the input valid time is 1.5 clock cycles, for slave the output valid time is 2.5 clock cycles. So if FlexIO SPI master communicates with other spi IPs, the maximum baud rate is FlexIO clock frequency divided by 2*2=4. If FlexIO SPI master communicates with FlexIO SPI slave, the maximum baud rate is FlexIO clock frequency divided by (1.5+2.5)*2=8.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- masterConfig – Pointer to the flexio_spi_master_config_t structure.
- srcClock_Hz – FlexIO source clock in Hz.

void FLEXIO_SPI_MasterDeinit(*FLEXIO_SPI_Type* *base)

Resets the FlexIO SPI timer and shifter config.

Parameters

- base – Pointer to the FLEXIO_SPI_Type.

void FLEXIO_SPI_MasterGetDefaultConfig(*flexio_spi_master_config_t* *masterConfig)

Gets the default configuration to configure the FlexIO SPI master. The configuration can be used directly by calling the FLEXIO_SPI_MasterConfigure(). Example:

```
flexio_spi_master_config_t masterConfig;
FLEXIO_SPI_MasterGetDefaultConfig(&masterConfig);
```

Parameters

- masterConfig – Pointer to the flexio_spi_master_config_t structure.

void FLEXIO_SPI_SlaveInit(*FLEXIO_SPI_Type* *base, *flexio_spi_slave_config_t* *slaveConfig)

Ungates the FlexIO clock, resets the FlexIO module, configures the FlexIO SPI slave hardware configuration, and configures the FlexIO SPI with FlexIO SPI slave configuration. The configuration structure can be filled by the user, or be set with default values by the FLEXIO_SPI_SlaveGetDefaultConfig().

Note: 1.Only one timer is needed in the FlexIO SPI slave. As a result, the second timer index is ignored. 2.FlexIO SPI slave only support CPOL = 0, which means clock inactive low. 3.For FlexIO SPI master, the input valid time is 1.5 clock cycles, for slave the output valid time is 2.5 clock cycles. So if FlexIO SPI slave communicates with other spi IPs, the maximum baud rate is FlexIO clock frequency divided by 3*2=6. If FlexIO SPI slave communicates with FlexIO SPI master, the maximum baud rate is FlexIO clock frequency divided by (1.5+2.5)*2=8. Example

```
FLEXIO_SPI_Type spiDev = {
.flexioBase = FLEXIO,
.SDOPinIndex = 0,
.SDIPinIndex = 1,
.SCKPinIndex = 2,
.CSnPinIndex = 3,
.shifterIndex = {0,1},
.timerIndex = {0}
};
flexio_spi_slave_config_t config = {
.enableSlave = true,
.enableInDoze = false,
.enableInDebug = true,
.enableFastAccess = false,
.phase = kFLEXIO_SPI_ClockPhaseFirstEdge,
.direction = kFLEXIO_SPI_MsbFirst,
.dataMode = kFLEXIO_SPI_8BitMode
};
FLEXIO_SPI_SlaveInit(&spiDev, &config);
```

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- slaveConfig – Pointer to the flexio_spi_slave_config_t structure.

void FLEXIO_SPI_SlaveDeinit(*FLEXIO_SPI_Type* *base)

Gates the FlexIO clock.

Parameters

- base – Pointer to the FLEXIO_SPI_Type.

void FLEXIO_SPI_SlaveGetDefaultConfig(*flexio_spi_slave_config_t* *slaveConfig)

Gets the default configuration to configure the FlexIO SPI slave. The configuration can be used directly for calling the FLEXIO_SPI_SlaveConfigure(). Example:

```
flexio_spi_slave_config_t slaveConfig;
FLEXIO_SPI_SlaveGetDefaultConfig(&slaveConfig);
```

Parameters

- slaveConfig – Pointer to the flexio_spi_slave_config_t structure.

uint32_t FLEXIO_SPI_GetStatusFlags(*FLEXIO_SPI_Type* *base)

Gets FlexIO SPI status flags.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.

Returns

status flag; Use the status flag to AND the following flag mask and get the status.

- kFLEXIO_SPI_TxEmptyFlag
- kFLEXIO_SPI_RxEmptyFlag

void FLEXIO_SPI_ClearStatusFlags(*FLEXIO_SPI_Type* *base, uint32_t mask)

Clears FlexIO SPI status flags.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – status flag The parameter can be any combination of the following values:
  - kFLEXIO_SPI_TxEmptyFlag
  - kFLEXIO_SPI_RxEmptyFlag

void FLEXIO_SPI_EnableInterrupts(*FLEXIO_SPI_Type* *base, uint32_t mask)

Enables the FlexIO SPI interrupt.

This function enables the FlexIO SPI interrupt.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – interrupt source. The parameter can be any combination of the following values:
  - kFLEXIO_SPI_RxFullInterruptEnable
  - kFLEXIO_SPI_TxEmptyInterruptEnable

void FLEXIO_SPI_DisableInterrupts(*FLEXIO_SPI_Type* *base, uint32_t mask)

Disables the FlexIO SPI interrupt.

This function disables the FlexIO SPI interrupt.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – interrupt source The parameter can be any combination of the following values:
  - kFLEXIO_SPI_RxFullInterruptEnable
  - kFLEXIO_SPI_TxEmptyInterruptEnable

void FLEXIO_SPI_EnableDMA(*FLEXIO_SPI_Type* *base, uint32_t mask, bool enable)

Enables/disables the FlexIO SPI transmit DMA. This function enables/disables the FlexIO SPI Tx DMA, which means that asserting the kFLEXIO_SPI_TxEmptyFlag does/doesn't trigger the DMA request.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- mask – SPI DMA source.
- enable – True means enable DMA, false means disable DMA.

static inline uint32_t FLEXIO_SPI_GetTxDataRegisterAddress(*FLEXIO_SPI_Type* *base, *flexio_spi_shift_direction_t* direction)

Gets the FlexIO SPI transmit data register address for MSB first transfer.

This function returns the SPI data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.

Returns
FlexIO SPI transmit data register address.

static inline uint32_t FLEXIO_SPI_GetRxDataRegisterAddress(*FLEXIO_SPI_Type* *base, *flexio_spi_shift_direction_t* direction)

Gets the FlexIO SPI receive data register address for the MSB first transfer.

This function returns the SPI data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.

Returns
FlexIO SPI receive data register address.

static inline void FLEXIO_SPI_Enable(*FLEXIO_SPI_Type* *base, bool enable)

Enables/disables the FlexIO SPI module operation.

Parameters

- base – Pointer to the FLEXIO_SPI_Type.
- enable – True to enable, false does not have any effect.

void FLEXIO_SPI_MasterSetBaudRate(*FLEXIO_SPI_Type* *base, uint32_t baudRate_Bps, uint32_t srcClockHz)

Sets baud rate for the FlexIO SPI transfer, which is only used for the master.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- baudRate_Bps – Baud Rate needed in Hz.
- srcClockHz – SPI source clock frequency in Hz.

static inline void FLEXIO_SPI_WriteData(*FLEXIO_SPI_Type* *base, *flexio_spi_shift_direction_t* direction, uint32_t data)

Writes one byte of data, which is sent using the MSB method.

Note: This is a non-blocking API, which returns directly after the data is put into the data register but the data transfer is not finished on the bus. Ensure that the TxEmptyFlag is asserted before calling this API.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.
- data – 8/16/32 bit data.

static inline uint32_t FLEXIO_SPI_ReadData(*FLEXIO_SPI_Type* *base, *flexio_spi_shift_direction_t* direction)

Reads 8 bit/16 bit data.

Note: This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the RxFullFlag is asserted before calling this API.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.

Returns

8 bit/16 bit data received.

status_t FLEXIO_SPI_WriteBlocking(*FLEXIO_SPI_Type* *base, *flexio_spi_shift_direction_t* direction, const uint8_t *buffer, size_t size)

Sends a buffer of data bytes.

Note: This function blocks using the polling method until all bytes have been sent.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.
- buffer – The data bytes to send.
- size – The number of data bytes to send.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_FLEXIO_SPI_Timeout – The transfer timed out and was aborted.

status_t FLEXIO_SPI_ReadBlocking(*FLEXIO_SPI_Type* *base, *flexio_spi_shift_direction_t* direction, uint8_t *buffer, size_t size)

Receives a buffer of bytes.

Note: This function blocks using the polling method until all bytes have been received.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- direction – Shift direction of MSB first or LSB first.
- buffer – The buffer to store the received bytes.
- size – The number of data bytes to be received.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_FLEXIO_SPI_Timeout – The transfer timed out and was aborted.

status_t FLEXIO_SPI_MasterTransferBlocking(*FLEXIO_SPI_Type* *base, *flexio_spi_transfer_t* *xfer)

Receives a buffer of bytes.

Note: This function blocks via polling until all bytes have been received.

Parameters

- base – pointer to FLEXIO_SPI_Type structure
- xfer – FlexIO SPI transfer structure, see flexio_spi_transfer_t.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_FLEXIO_SPI_Timeout – The transfer timed out and was aborted.

void FLEXIO_SPI_FlushShifters(*FLEXIO_SPI_Type* *base)

Flush tx/rx shifters.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.

status_t FLEXIO_SPI_MasterTransferCreateHandle(*FLEXIO_SPI_Type* *base, *flexio_spi_master_handle_t* *handle, *flexio_spi_master_transfer_callback_t* callback, void *userData)

Initializes the FlexIO SPI Master handle, which is used in transactional functions.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_master_handle_t structure to store the transfer state.
- callback – The callback function.
- userData – The parameter of the callback function.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

status_t FLEXIO_SPI_MasterTransferNonBlocking(*FLEXIO_SPI_Type* *base, *flexio_spi_master_handle_t* *handle, *flexio_spi_transfer_t* *xfer)

Master transfer data using IRQ.

This function sends data using IRQ. This is a non-blocking function, which returns right away. When all data is sent out/received, the callback function is called.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_master_handle_t structure to store the transfer state.
- xfer – FlexIO SPI transfer structure. See flexio_spi_transfer_t.

Return values

- kStatus_Success – Successfully start a transfer.

- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_FLEXIO_SPI_Busy – SPI is not idle, is running another transfer.

void FLEXIO_SPI_MasterTransferAbort(*FLEXIO_SPI_Type* *base, *flexio_spi_master_handle_t* *handle)

Aborts the master data transfer, which used IRQ.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_master_handle_t structure to store the transfer state.

status_t FLEXIO_SPI_MasterTransferGetCount(*FLEXIO_SPI_Type* *base, *flexio_spi_master_handle_t* *handle, size_t *count)

Gets the data transfer status which used IRQ.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_master_handle_t structure to store the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_InvalidArgument – count is Invalid.
- kStatus_Success – Successfully return the count.

void FLEXIO_SPI_MasterTransferHandleIRQ(void *spiType, void *spiHandle)

FlexIO SPI master IRQ handler function.

Parameters

- spiType – Pointer to the FLEXIO_SPI_Type structure.
- spiHandle – Pointer to the flexio_spi_master_handle_t structure to store the transfer state.

status_t FLEXIO_SPI_SlaveTransferCreateHandle(*FLEXIO_SPI_Type* *base, *flexio_spi_slave_handle_t* *handle, *flexio_spi_slave_transfer_callback_t* callback, void *userData)

Initializes the FlexIO SPI Slave handle, which is used in transactional functions.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.
- callback – The callback function.
- userData – The parameter of the callback function.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

status_t FLEXIO_SPI_SlaveTransferNonBlocking(*FLEXIO_SPI_Type* *base, *flexio_spi_slave_handle_t* *handle, *flexio_spi_transfer_t* *xfer)

Slave transfer data using IRQ.

This function sends data using IRQ. This is a non-blocking function, which returns right away. When all data is sent out/received, the callback function is called.

Parameters

- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.
- base – Pointer to the FLEXIO_SPI_Type structure.
- xfer – FlexIO SPI transfer structure. See flexio_spi_transfer_t.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_FLEXIO_SPI_Busy – SPI is not idle; it is running another transfer.

static inline void FLEXIO_SPI_SlaveTransferAbort(*FLEXIO_SPI_Type* *base, *flexio_spi_slave_handle_t* *handle)

Aborts the slave data transfer which used IRQ, share same API with master.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.

static inline *status_t* FLEXIO_SPI_SlaveTransferGetCount(*FLEXIO_SPI_Type* *base, *flexio_spi_slave_handle_t* *handle, size_t *count)

Gets the data transfer status which used IRQ, share same API with master.

Parameters

- base – Pointer to the FLEXIO_SPI_Type structure.
- handle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_InvalidArgument – count is Invalid.
- kStatus_Success – Successfully return the count.

void FLEXIO_SPI_SlaveTransferHandleIRQ(void *spiType, void *spiHandle)

FlexIO SPI slave IRQ handler function.

Parameters

- spiType – Pointer to the FLEXIO_SPI_Type structure.
- spiHandle – Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.

FSL_FLEXIO_SPI_DRIVER_VERSION

FlexIO SPI driver version.

Error codes for the FlexIO SPI driver.

Values:

enumerator kStatus_FLEXIO_SPI_Busy
FlexIO SPI is busy.

enumerator kStatus_FLEXIO_SPI_Idle
SPI is idle

enumerator kStatus_FLEXIO_SPI_Error
FlexIO SPI error.

enumerator kStatus_FLEXIO_SPI_Timeout
FlexIO SPI timeout polling status flags.

enum _flexio_spi_clock_phase
FlexIO SPI clock phase configuration.

Values:

enumerator kFLEXIO_SPI_ClockPhaseFirstEdge
First edge on SPSCK occurs at the middle of the first cycle of a data transfer.

enumerator kFLEXIO_SPI_ClockPhaseSecondEdge
First edge on SPSCK occurs at the start of the first cycle of a data transfer.

enum _flexio_spi_shift_direction
FlexIO SPI data shifter direction options.

Values:

enumerator kFLEXIO_SPI_MsbFirst
Data transfers start with most significant bit.

enumerator kFLEXIO_SPI_LsbFirst
Data transfers start with least significant bit.

enum _flexio_spi_data_bitcount_mode
FlexIO SPI data length mode options.

Values:

enumerator kFLEXIO_SPI_8BitMode
8-bit data transmission mode.

enumerator kFLEXIO_SPI_16BitMode
16-bit data transmission mode.

enumerator kFLEXIO_SPI_32BitMode
32-bit data transmission mode.

enum _flexio_spi_interrupt_enable
Define FlexIO SPI interrupt mask.

Values:

enumerator kFLEXIO_SPI_TxEmptyInterruptEnable
Transmit buffer empty interrupt enable.

enumerator kFLEXIO_SPI_RxFullInterruptEnable
Receive buffer full interrupt enable.

enum _flexio_spi_status_flags

Define FlexIO SPI status mask.

Values:

enumerator kFLEXIO_SPI_TxBufferEmptyFlag

Transmit buffer empty flag.

enumerator kFLEXIO_SPI_RxBufferFullFlag

Receive buffer full flag.

enum _flexio_spi_dma_enable

Define FlexIO SPI DMA mask.

Values:

enumerator kFLEXIO_SPI_TxDmaEnable

Tx DMA request source

enumerator kFLEXIO_SPI_RxDmaEnable

Rx DMA request source

enumerator kFLEXIO_SPI_DmaAllEnable

All DMA request source

enum _flexio_spi_transfer_flags

Define FlexIO SPI transfer flags.

Note: Use kFLEXIO_SPI_csContinuous and one of the other flags to OR together to form the transfer flag.

Values:

enumerator kFLEXIO_SPI_8bitMsb

FlexIO SPI 8-bit MSB first

enumerator kFLEXIO_SPI_8bitLsb

FlexIO SPI 8-bit LSB first

enumerator kFLEXIO_SPI_16bitMsb

FlexIO SPI 16-bit MSB first

enumerator kFLEXIO_SPI_16bitLsb

FlexIO SPI 16-bit LSB first

enumerator kFLEXIO_SPI_32bitMsb

FlexIO SPI 32-bit MSB first

enumerator kFLEXIO_SPI_32bitLsb

FlexIO SPI 32-bit LSB first

enumerator kFLEXIO_SPI_csContinuous

Enable the CS signal continuous mode

typedef enum *_flexio_spi_clock_phase* flexio_spi_clock_phase_t

FlexIO SPI clock phase configuration.

typedef enum *_flexio_spi_shift_direction* flexio_spi_shift_direction_t

FlexIO SPI data shifter direction options.

typedef enum *_flexio_spi_data_bitcount_mode* flexio_spi_data_bitcount_mode_t

FlexIO SPI data length mode options.

typedef struct *_flexio_spi_type* FLEXIO_SPI_Type

Define FlexIO SPI access structure typedef.

typedef struct *_flexio_spi_master_config* flexio_spi_master_config_t

Define FlexIO SPI master configuration structure.

typedef struct *_flexio_spi_slave_config* flexio_spi_slave_config_t

Define FlexIO SPI slave configuration structure.

typedef struct *_flexio_spi_transfer* flexio_spi_transfer_t

Define FlexIO SPI transfer structure.

typedef struct *_flexio_spi_master_handle* flexio_spi_master_handle_t

typedef for flexio_spi_master_handle_t in advance.

typedef *flexio_spi_master_handle_t* flexio_spi_slave_handle_t

Slave handle is the same with master handle.

typedef void (*flexio_spi_master_transfer_callback_t)(*FLEXIO_SPI_Type* *base, *flexio_spi_master_handle_t* *handle, *status_t* status, void *userData)

FlexIO SPI master callback for finished transmit.

typedef void (*flexio_spi_slave_transfer_callback_t)(*FLEXIO_SPI_Type* *base, *flexio_spi_slave_handle_t* *handle, *status_t* status, void *userData)

FlexIO SPI slave callback for finished transmit.

FLEXIO_SPI_DUMMYDATA

FlexIO SPI dummy transfer data, the data is sent while txData is NULL.

SPI_RETRY_TIMES

Retry times for waiting flag.

FLEXIO_SPI_XFER_DATA_FORMAT(flag)

Get the transfer data format of width and bit order.

struct _flexio_spi_type

#include <fsl_flexio_spi.h> Define FlexIO SPI access structure typedef.

Public Members

FLEXIO_Type *flexioBase

FlexIO base pointer.

uint8_t SDOPinIndex

Pin select for data output. To set SDO pin in Hi-Z state, user needs to mux the pin as GPIO input and disable all pull up/down in application.

uint8_t SDIPinIndex

Pin select for data input.

uint8_t SCKPinIndex

Pin select for clock.

uint8_t CSnPinIndex

Pin select for enable.

uint8_t shifterIndex[2]

Shifter index used in FlexIO SPI.

uint8_t timerIndex[2]

Timer index used in FlexIO SPI.

struct __flexio_spi_master_config

#include <fsl_flexio_spi.h> Define FlexIO SPI master configuration structure.

Public Members

bool enableMaster

Enable/disable FlexIO SPI master after configuration.

bool enableInDoze

Enable/disable FlexIO operation in doze mode.

bool enableInDebug

Enable/disable FlexIO operation in debug mode.

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

uint32_t baudRate_Bps

Baud rate in Bps.

flexio_spi_clock_phase_t phase

Clock phase.

flexio_spi_data_bitcount_mode_t dataMode

8bit or 16bit mode.

struct __flexio_spi_slave_config

#include <fsl_flexio_spi.h> Define FlexIO SPI slave configuration structure.

Public Members

bool enableSlave

Enable/disable FlexIO SPI slave after configuration.

bool enableInDoze

Enable/disable FlexIO operation in doze mode.

bool enableInDebug

Enable/disable FlexIO operation in debug mode.

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

flexio_spi_clock_phase_t phase

Clock phase.

flexio_spi_data_bitcount_mode_t dataMode

8bit or 16bit mode.

struct __flexio_spi_transfer

#include <fsl_flexio_spi.h> Define FlexIO SPI transfer structure.

Public Members

const uint8_t *txData

Send buffer.

uint8_t *rxData
Receive buffer.

size_t dataSize
Transfer bytes.

uint8_t flags
FlexIO SPI control flag, MSB first or LSB first.

struct __flexio_spi_master_handle
#include <fsl_flexio_spi.h> Define FlexIO SPI handle structure.

Public Members

const uint8_t *txData
Transfer buffer.

uint8_t *rxData
Receive buffer.

size_t transferSize
Total bytes to be transferred.

volatile size_t txRemainingBytes
Send data remaining in bytes.

volatile size_t rxRemainingBytes
Receive data remaining in bytes.

volatile uint32_t state
FlexIO SPI internal state.

uint8_t bytePerFrame
SPI mode, 2bytes or 1byte in a frame

flexio_spi_shift_direction_t direction
Shift direction.

flexio_spi_master_transfer_callback_t callback
FlexIO SPI callback.

void *userData
Callback parameter.

bool isCsContinuous
Is current transfer using CS continuous mode.

uint32_t timer1Cfg
TIMER1 TIMCFG regiser value backup.

2.21 FlexIO UART Driver

status_t FLEXIO_UART_Init(*FLEXIO_UART_Type* *base, const *flexio_uart_config_t* *userConfig, uint32_t srcClock_Hz)

Ungates the FlexIO clock, resets the FlexIO module, configures FlexIO UART hardware, and configures the FlexIO UART with FlexIO UART configuration. The configuration structure can be filled by the user or be set with default values by FLEXIO_UART_GetDefaultConfig().

Example

```
FLEXIO_UART_Type base = {
.flexioBase = FLEXIO,
.TxPinIndex = 0,
.RxPinIndex = 1,
.shifterIndex = {0,1},
.timerIndex = {0,1}
};
flexio_uart_config_t config = {
.enableInDoze = false,
.enableInDebug = true,
.enableFastAccess = false,
.baudRate_Bps = 115200U,
.bitCountPerChar = 8
};
FLEXIO_UART_Init(base, &config, srcClock_Hz);
```

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- userConfig – Pointer to the flexio_uart_config_t structure.
- srcClock_Hz – FlexIO source clock in Hz.

Return values

- kStatus_Success – Configuration success.
- kStatus_FLEXIO_UART_BaudrateNotSupport – Baudrate is not supported for current clock source frequency.

void FLEXIO_UART_Deinit(*FLEXIO_UART_Type* *base)

Resets the FlexIO UART shifter and timer config.

Note: After calling this API, call the FLEXO_UART_Init to use the FlexIO UART module.

Parameters

- base – Pointer to FLEXIO_UART_Type structure

void FLEXIO_UART_GetDefaultConfig(*flexio_uart_config_t* *userConfig)

Gets the default configuration to configure the FlexIO UART. The configuration can be used directly for calling the FLEXIO_UART_Init(). Example:

```
flexio_uart_config_t config;
FLEXIO_UART_GetDefaultConfig(&userConfig);
```

Parameters

- userConfig – Pointer to the flexio_uart_config_t structure.

uint32_t FLEXIO_UART_GetStatusFlags(*FLEXIO_UART_Type* *base)

Gets the FlexIO UART status flags.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.

Returns

FlexIO UART status flags.

void FLEXIO_UART_ClearStatusFlags(*FLEXIO_UART_Type* *base, uint32_t mask)

Gets the FlexIO UART status flags.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- mask – Status flag. The parameter can be any combination of the following values:
  - kFLEXIO_UART_TxDataRegEmptyFlag
  - kFLEXIO_UART_RxEmptyFlag
  - kFLEXIO_UART_RxOverRunFlag

void FLEXIO_UART_EnableInterrupts(*FLEXIO_UART_Type* *base, uint32_t mask)

Enables the FlexIO UART interrupt.

This function enables the FlexIO UART interrupt.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- mask – Interrupt source.

void FLEXIO_UART_DisableInterrupts(*FLEXIO_UART_Type* *base, uint32_t mask)

Disables the FlexIO UART interrupt.

This function disables the FlexIO UART interrupt.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- mask – Interrupt source.

static inline uint32_t FLEXIO_UART_GetTxDataRegisterAddress(*FLEXIO_UART_Type* *base)

Gets the FlexIO UARt transmit data register address.

This function returns the UART data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.

Returns

FlexIO UART transmit data register address.

static inline uint32_t FLEXIO_UART_GetRxDataRegisterAddress(*FLEXIO_UART_Type* *base)

Gets the FlexIO UART receive data register address.

This function returns the UART data register address, which is mainly used by DMA/eDMA.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.

Returns

FlexIO UART receive data register address.

static inline void FLEXIO_UART_EnableTxDMA(*FLEXIO_UART_Type* *base, bool enable)

Enables/disables the FlexIO UART transmit DMA. This function enables/disables the FlexIO UART Tx DMA, which means asserting the kFLEXIO_UART_TxDataRegEmptyFlag does/doesn't trigger the DMA request.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- enable – True to enable, false to disable.

static inline void FLEXIO_UART_EnableRxDMA(*FLEXIO_UART_Type* *base, bool enable)

Enables/disables the FlexIO UART receive DMA. This function enables/disables the FlexIO UART Rx DMA, which means asserting kFLEXIO_UART_RxDataRegFullFlag does/doesn't trigger the DMA request.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- enable – True to enable, false to disable.

static inline void FLEXIO_UART_Enable(*FLEXIO_UART_Type* *base, bool enable)

Enables/disables the FlexIO UART module operation.

Parameters

- base – Pointer to the FLEXIO_UART_Type.
- enable – True to enable, false does not have any effect.

static inline void FLEXIO_UART_WriteByte(*FLEXIO_UART_Type* *base, const uint8_t *buffer)

Writes one byte of data.

Note: This is a non-blocking API, which returns directly after the data is put into the data register. Ensure that the TxEmptyFlag is asserted before calling this API.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- buffer – The data bytes to send.

static inline void FLEXIO_UART_ReadByte(*FLEXIO_UART_Type* *base, uint8_t *buffer)

Reads one byte of data.

Note: This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the RxFullFlag is asserted before calling this API.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- buffer – The buffer to store the received bytes.

status_t FLEXIO_UART_WriteBlocking(*FLEXIO_UART_Type* *base, const uint8_t *txData, size_t txSize)

Sends a buffer of data bytes.

Note: This function blocks using the polling method until all bytes have been sent.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- txData – The data bytes to send.
- txSize – The number of data bytes to send.

Return values

- kStatus_FLEXIO_UART_Timeout – Transmission timed out and was aborted.

- kStatus_Success – Successfully wrote all data.

status_t FLEXIO_UART_ReadBlocking(*FLEXIO_UART_Type* *base, uint8_t *rxData, size_t rxSize)

Receives a buffer of bytes.

Note: This function blocks using the polling method until all bytes have been received.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- rxData – The buffer to store the received bytes.
- rxSize – The number of data bytes to be received.

Return values

- kStatus_FLEXIO_UART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully received all data.

status_t FLEXIO_UART_TransferCreateHandle(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle, *flexio_uart_transfer_callback_t* callback, void *userData)

Initializes the UART handle.

This function initializes the FlexIO UART handle, which can be used for other FlexIO UART transactional APIs. Call this API once to get the initialized handle.

The UART driver supports the "background" receiving, which means that users can set up a RX ring buffer optionally. Data received is stored into the ring buffer even when the user doesn't call the FLEXIO_UART_TransferReceiveNonBlocking() API. If there is already data received in the ring buffer, users can get the received data from the ring buffer directly. The ring buffer is disabled if passing NULL as ringBuffer.

Parameters

- base – to FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.
- callback – The callback function.
- userData – The parameter of the callback function.

Return values

- kStatus_Success – Successfully create the handle.
- kStatus_OutOfRange – The FlexIO type/handle/ISR table out of range.

void FLEXIO_UART_TransferStartRingBuffer(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle, uint8_t *ringBuffer, size_t ringBufferSize)

Sets up the RX ring buffer.

This function sets up the RX ring buffer to a specific UART handle.

When the RX ring buffer is used, data received is stored into the ring buffer even when the user doesn't call the UART_ReceiveNonBlocking() API. If there is already data received in the ring buffer, users can get the received data from the ring buffer directly.

Note: When using the RX ring buffer, one byte is reserved for internal use. In other words, if ringBufferSize is 32, only 31 bytes are used for saving data.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.
- ringBuffer – Start address of ring buffer for background receiving. Pass NULL to disable the ring buffer.
- ringBufferSize – Size of the ring buffer.

void FLEXIO_UART_TransferStopRingBuffer(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle)

Aborts the background transfer and uninstalls the ring buffer.

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.

status_t FLEXIO_UART_TransferSendNonBlocking(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle, *flexio_uart_transfer_t* *xfer)

Transmits a buffer of data using the interrupt method.

This function sends data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data to be written to the TX register. When all data is written to the TX register in ISR, the FlexIO UART driver calls the callback function and passes the kStatus_FLEXIO_UART_TxIdle as status parameter.

Note: The kStatus_FLEXIO_UART_TxIdle is passed to the upper layer when all data is written to the TX register. However, it does not ensure that all data is sent out.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.
- xfer – FlexIO UART transfer structure. See flexio_uart_transfer_t.

Return values

- kStatus_Success – Successfully starts the data transmission.
- kStatus_UART_TxBusy – Previous transmission still not finished, data not written to the TX register.

void FLEXIO_UART_TransferAbortSend(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle)

Aborts the interrupt-driven data transmit.

This function aborts the interrupt-driven data sending. Get the remainBytes to find out how many bytes are still not sent out.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.

status_t FLEXIO_UART_TransferGetSendCount(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle, size_t *count)

Gets the number of bytes sent.

This function gets the number of bytes sent driven by interrupt.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.
- count – Number of bytes sent so far by the non-blocking transaction.

Return values

- kStatus_NoTransferInProgress – transfer has finished or no transfer in progress.
- kStatus_Success – Successfully return the count.

status_t FLEXIO_UART_TransferReceiveNonBlocking(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle, *flexio_uart_transfer_t* *xfer, size_t *receivedBytes)

Receives a buffer of data using the interrupt method.

This function receives data using the interrupt method. This is a non-blocking function, which returns without waiting for all data to be received. If the RX ring buffer is used and not empty, the data in ring buffer is copied and the parameter receivedBytes shows how many bytes are copied from the ring buffer. After copying, if the data in ring buffer is not enough to read, the receive request is saved by the UART driver. When new data arrives, the receive request is serviced first. When all data is received, the UART driver notifies the upper layer through a callback function and passes the status parameter kStatus_UART_RxIdle. For example, if the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer, the 5 bytes are copied to xfer->data. This function returns with the parameter receivedBytes set to 5. For the last 5 bytes, newly arrived data is saved from the xfer->data[5]. When 5 bytes are received, the UART driver notifies upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to xfer->data. When all data is received, the upper layer is notified.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.
- xfer – UART transfer structure. See flexio_uart_transfer_t.
- receivedBytes – Bytes received from the ring buffer directly.

Return values

- kStatus_Success – Successfully queue the transfer into the transmit queue.
- kStatus_FLEXIO_UART_RxBusy – Previous receive request is not finished.

void FLEXIO_UART_TransferAbortReceive(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle)

Aborts the receive data which was using IRQ.

This function aborts the receive data which was using IRQ.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.

status_t FLEXIO_UART_TransferGetReceiveCount(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle, size_t *count)

Gets the number of bytes received.

This function gets the number of bytes received driven by interrupt.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.
- handle – Pointer to the flexio_uart_handle_t structure to store the transfer state.
- count – Number of bytes received so far by the non-blocking transaction.

Return values

- kStatus_NoTransferInProgress – transfer has finished or no transfer in progress.
- kStatus_Success – Successfully return the count.

void FLEXIO_UART_TransferHandleIRQ(void *uartType, void *uartHandle)

FlexIO UART IRQ handler function.

This function processes the FlexIO UART transmit and receives the IRQ request.

Parameters

- uartType – Pointer to the FLEXIO_UART_Type structure.
- uartHandle – Pointer to the flexio_uart_handle_t structure to store the transfer state.

void FLEXIO_UART_FlushShifters(*FLEXIO_UART_Type* *base)

Flush tx/rx shifters.

Parameters

- base – Pointer to the FLEXIO_UART_Type structure.

FSL_FLEXIO_UART_DRIVER_VERSION

FlexIO UART driver version.

Error codes for the UART driver.

Values:

enumerator kStatus_FLEXIO_UART_TxBusy

Transmitter is busy.

enumerator kStatus_FLEXIO_UART_RxBusy

Receiver is busy.

enumerator kStatus_FLEXIO_UART_TxIdle
UART transmitter is idle.

enumerator kStatus_FLEXIO_UART_RxIdle
UART receiver is idle.

enumerator kStatus_FLEXIO_UART_ERROR
ERROR happens on UART.

enumerator kStatus_FLEXIO_UART_RxRingBufferOverrun
UART RX software ring buffer overrun.

enumerator kStatus_FLEXIO_UART_RxHardwareOverrun
UART RX receiver overrun.

enumerator kStatus_FLEXIO_UART_Timeout
UART times out.

enumerator kStatus_FLEXIO_UART_BaudrateNotSupport
Baudrate is not supported in current clock source

enum _flexio_uart_bit_count_per_char
FlexIO UART bit count per char.

Values:

enumerator kFLEXIO_UART_7BitsPerChar
7-bit data characters

enumerator kFLEXIO_UART_8BitsPerChar
8-bit data characters

enumerator kFLEXIO_UART_9BitsPerChar
9-bit data characters

enum _flexio_uart_interrupt_enable
Define FlexIO UART interrupt mask.

Values:

enumerator kFLEXIO_UART_TxDataRegEmptyInterruptEnable
Transmit buffer empty interrupt enable.

enumerator kFLEXIO_UART_RxDataRegFullInterruptEnable
Receive buffer full interrupt enable.

enum _flexio_uart_status_flags
Define FlexIO UART status mask.

Values:

enumerator kFLEXIO_UART_TxDataRegEmptyFlag
Transmit buffer empty flag.

enumerator kFLEXIO_UART_RxDataRegFullFlag
Receive buffer full flag.

enumerator kFLEXIO_UART_RxOverRunFlag
Receive buffer over run flag.

typedef enum *_flexio_uart_bit_count_per_char* flexio_uart_bit_count_per_char_t
FlexIO UART bit count per char.

typedef struct *_flexio_uart_type* FLEXIO_UART_Type

Define FlexIO UART access structure typedef.

typedef struct *_flexio_uart_config* flexio_uart_config_t

Define FlexIO UART user configuration structure.

typedef struct *_flexio_uart_transfer* flexio_uart_transfer_t

Define FlexIO UART transfer structure.

typedef struct *_flexio_uart_handle* flexio_uart_handle_t

typedef void (*flexio_uart_transfer_callback_t)(*FLEXIO_UART_Type* *base, *flexio_uart_handle_t* *handle, *status_t* status, void *userData)

FlexIO UART transfer callback function.

UART_RETRY_TIMES

Retry times for waiting flag.

struct _flexio_uart_type

#include <fsl_flexio_uart.h> Define FlexIO UART access structure typedef.

Public Members

FLEXIO_Type *flexioBase

FlexIO base pointer.

uint8_t TxPinIndex

Pin select for UART_Tx.

uint8_t RxPinIndex

Pin select for UART_Rx.

uint8_t shifterIndex[2]

Shifter index used in FlexIO UART.

uint8_t timerIndex[2]

Timer index used in FlexIO UART.

struct _flexio_uart_config

#include <fsl_flexio_uart.h> Define FlexIO UART user configuration structure.

Public Members

bool enableUart

Enable/disable FlexIO UART TX & RX.

bool enableInDoze

Enable/disable FlexIO operation in doze mode

bool enableInDebug

Enable/disable FlexIO operation in debug mode

bool enableFastAccess

Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

uint32_t baudRate_Bps

Baud rate in Bps.

flexio_uart_bit_count_per_char_t bitCountPerChar
number of bits, 7/8/9 -bit

struct __flexio_uart__transfer
#include <fsl_flexio_uart.h> Define FlexIO UART transfer structure.

Public Members

size_t dataSize
Transfer size

struct __flexio_uart__handle
#include <fsl_flexio_uart.h> Define FLEXIO UART handle structure.

Public Members

const uint8_t *volatile txData
Address of remaining data to send.

volatile size_t txDataSize
Size of the remaining data to send.

uint8_t *volatile rxData
Address of remaining data to receive.

volatile size_t rxDataSize
Size of the remaining data to receive.

size_t txDataSizeAll
Total bytes to be sent.

size_t rxDataSizeAll
Total bytes to be received.

uint8_t *rxRingBuffer
Start address of the receiver ring buffer.

size_t rxRingBufferSize
Size of the ring buffer.

volatile uint16_t rxRingBufferHead
Index for the driver to store received data into ring buffer.

volatile uint16_t rxRingBufferTail
Index for the user to get data from the ring buffer.

flexio_uart_transfer_callback_t callback
Callback function.

void *userData
UART callback function parameter.

volatile uint8_t txState
TX transfer state.

volatile uint8_t rxState
RX transfer state

union ___unnamed78___

Public Members

uint8_t *data
The buffer of data to be transfer.

uint8_t *rxData
The buffer to receive data.

const uint8_t *txData
The buffer of data to be sent.

2.22 GPIO: General-Purpose Input/Output Driver

FSL_GPIO_DRIVER_VERSION
GPIO driver version.

enum _gpio_pin_direction
GPIO direction definition.

Values:

enumerator kGPIO_DigitalInput
Set current pin as digital input

enumerator kGPIO_DigitalOutput
Set current pin as digital output

enum _gpio_checker_attribute
GPIO checker attribute.

Values:

enumerator kGPIO_UsernonsecureRWUsersecureRWPrivilegedsecureRW
User nonsecure:Read+Write; User Secure:Read+Write; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureRUsersecureRWPrivilegedsecureRW
User nonsecure:Read; User Secure:Read+Write; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureNUsersecureRWPrivilegedsecureRW
User nonsecure:None; User Secure:Read+Write; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureRUsersecureRPrivilegedsecureRW
User nonsecure:Read; User Secure:Read; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureNUsersecureRPrivilegedsecureRW
User nonsecure:None; User Secure:Read; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureNUsersecureNPrivilegedsecureRW
User nonsecure:None; User Secure:None; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureNUsersecureNPrivilegedsecureR
User nonsecure:None; User Secure:None; Privileged Secure:Read

enumerator kGPIO_UsernonsecureNUsersecureNPrivilegedsecureN
User nonsecure:None; User Secure:None; Privileged Secure:None

enumerator kGPIO_IgnoreAttributeCheck
Ignores the attribute check

enum _gpio_interrupt_config

Configures the interrupt generation condition.

Values:

enumerator kGPIO_InterruptStatusFlagDisabled

Interrupt status flag is disabled.

enumerator kGPIO_DMARisingEdge

ISF flag and DMA request on rising edge.

enumerator kGPIO_DMAFallingEdge

ISF flag and DMA request on falling edge.

enumerator kGPIO_DMAEitherEdge

ISF flag and DMA request on either edge.

enumerator kGPIO_FlagRisingEdge

Flag sets on rising edge.

enumerator kGPIO_FlagFallingEdge

Flag sets on falling edge.

enumerator kGPIO_FlagEitherEdge

Flag sets on either edge.

enumerator kGPIO_InterruptLogicZero

Interrupt when logic zero.

enumerator kGPIO_InterruptRisingEdge

Interrupt on rising edge.

enumerator kGPIO_InterruptFallingEdge

Interrupt on falling edge.

enumerator kGPIO_InterruptEitherEdge

Interrupt on either edge.

enumerator kGPIO_InterruptLogicOne

Interrupt when logic one.

enumerator kGPIO_ActiveHighTriggerOutputEnable

Enable active high-trigger output.

enumerator kGPIO_ActiveLowTriggerOutputEnable

Enable active low-trigger output.

enum _gpio_interrupt_selection

Configures the selection of interrupt/DMA request/trigger output.

Values:

enumerator kGPIO_InterruptOutput0

Interrupt/DMA request/trigger output 0.

enumerator kGPIO_InterruptOutput1

Interrupt/DMA request/trigger output 1.

enum gpio_pin_interrupt_control_t

GPIO pin and interrupt control.

Values:

enumerator kGPIO_PinControlNonSecure
Pin Control Non-Secure.

enumerator kGPIO_InterruptControlNonSecure
Interrupt Control Non-Secure.

enumerator kGPIO_PinControlNonPrivilege
Pin Control Non-Privilege.

enumerator kGPIO_InterruptControlNonPrivilege
Interrupt Control Non-Privilege.

typedef enum *_gpio_pin_direction* gpio_pin_direction_t
GPIO direction definition.

typedef enum *_gpio_checker_attribute* gpio_checker_attribute_t
GPIO checker attribute.

typedef struct *_gpio_pin_config* gpio_pin_config_t
The GPIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, leave the outputConfig unused. Note that in some use cases, the corresponding port property should be configured in advance with the PORT_SetPinConfig().

typedef enum *_gpio_interrupt_config* gpio_interrupt_config_t
Configures the interrupt generation condition.

typedef enum *_gpio_interrupt_selection* gpio_interrupt_selection_t
Configures the selection of interrupt/DMA request/trigger output.

typedef struct *_gpio_version_info* gpio_version_info_t
GPIO version information.

GPIO_FIT_REG(value)

struct _gpio_pin_config
#include <fsl_gpio.h> The GPIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, leave the outputConfig unused. Note that in some use cases, the corresponding port property should be configured in advance with the PORT_SetPinConfig().

Public Members

gpio_pin_direction_t pinDirection
GPIO direction, input or output

uint8_t outputLogic
Set a default output logic, which has no use in input

struct _gpio_version_info
#include <fsl_gpio.h> GPIO version information.

Public Members

uint16_t feature
Feature Specification Number.

uint8_t minor
Minor Version Number.

uint8_t major
Major Version Number.

2.23 GPIO Driver

void GPIO_PortInit(GPIO_Type *base)
Initializes the GPIO peripheral.

This function ungates the GPIO clock.

Parameters

- base – GPIO peripheral base pointer.

void GPIO_PortDenit(GPIO_Type *base)
Denitializes the GPIO peripheral.

Parameters

- base – GPIO peripheral base pointer.

void GPIO_PinInit(GPIO_Type *base, uint32_t pin, const *gpio_pin_config_t* *config)
Initializes a GPIO pin used by the board.

To initialize the GPIO, define a pin configuration, as either input or output, in the user file. Then, call the GPIO_PinInit() function.

This is an example to define an input pin or an output pin configuration.

```
Define a digital input pin configuration,
gpio_pin_config_t config =
{
  kGPIO_DigitalInput,
  0,
}
Define a digital output pin configuration,
gpio_pin_config_t config =
{
  kGPIO_DigitalOutput,
  0,
}
```

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- pin – GPIO port pin number
- config – GPIO pin configuration pointer

void GPIO_GetVersionInfo(GPIO_Type *base, *gpio_version_info_t* *verInfo)
Get GPIO version information.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- verInfo – GPIO version information

static inline void GPIO_SecurePrivilegeLock(GPIO_Type *base, *gpio_pin_interrupt_control_t* mask)
lock or unlock secure privilege.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – pin or interrupt macro

static inline void GPIO_EnablePinControlNonSecure(GPIO_Type *base, uint32_t mask)
Enable Pin Control Non-Secure.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_DisablePinControlNonSecure(GPIO_Type *base, uint32_t mask)
Disable Pin Control Non-Secure.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_EnablePinControlNonPrivilege(GPIO_Type *base, uint32_t mask)
Enable Pin Control Non-Privilege.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_DisablePinControlNonPrivilege(GPIO_Type *base, uint32_t mask)
Disable Pin Control Non-Privilege.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_EnableInterruptControlNonSecure(GPIO_Type *base, uint32_t mask)
Enable Interrupt Control Non-Secure.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_DisableInterruptControlNonSecure(GPIO_Type *base, uint32_t mask)
Disable Interrupt Control Non-Secure.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_EnableInterruptControlNonPrivilege(GPIO_Type *base, uint32_t mask)
Enable Interrupt Control Non-Privilege.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_DisableInterruptControlNonPrivilege(GPIO_Type *base, uint32_t mask)
Disable Interrupt Control Non-Privilege.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_PortInputEnable(GPIO_Type *base, uint32_t mask)

Enable port input.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_PortInputDisable(GPIO_Type *base, uint32_t mask)

Disable port input.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_PinWrite(GPIO_Type *base, uint32_t pin, uint8_t output)

Sets the output level of the multiple GPIO pins to the logic 1 or 0.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- pin – GPIO pin number
- output – GPIO pin output logic level.
  - 0: corresponding pin output low-logic level.
  - 1: corresponding pin output high-logic level.

static inline void GPIO_PortSet(GPIO_Type *base, uint32_t mask)

Sets the output level of the multiple GPIO pins to the logic 1.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_PortClear(GPIO_Type *base, uint32_t mask)

Sets the output level of the multiple GPIO pins to the logic 0.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_PortToggle(GPIO_Type *base, uint32_t mask)

Reverses the current output logic of the multiple GPIO pins.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline uint32_t GPIO_PinRead(GPIO_Type *base, uint32_t pin)

Reads the current input value of the GPIO port.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- pin – GPIO pin number

Return values

GPIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

static inline void GPIO_SetPinInterruptConfig(GPIO_Type *base, uint32_t pin, *gpio_interrupt_config_t* config)

Configures the gpio pin interrupt/DMA request.

Parameters

- base – GPIO peripheral base pointer.
- pin – GPIO pin number.
- config – GPIO pin interrupt configuration.
  - kGPIO_InterruptStatusFlagDisabled: Interrupt/DMA request disabled.
  - kGPIO_DMARisingEdge : DMA request on rising edge(if the DMA requests exit).
  - kGPIO_DMAFallingEdge: DMA request on falling edge(if the DMA requests exit).
  - kGPIO_DMAEitherEdge : DMA request on either edge(if the DMA requests exit).
  - kGPIO_FlagRisingEdge : Flag sets on rising edge(if the Flag states exit).
  - kGPIO_FlagFallingEdge : Flag sets on falling edge(if the Flag states exit).
  - kGPIO_FlagEitherEdge : Flag sets on either edge(if the Flag states exit).
  - kGPIO_InterruptLogicZero : Interrupt when logic zero.
  - kGPIO_InterruptRisingEdge : Interrupt on rising edge.
  - kGPIO_InterruptFallingEdge: Interrupt on falling edge.
  - kGPIO_InterruptEitherEdge : Interrupt on either edge.
  - kGPIO_InterruptLogicOne : Interrupt when logic one.
  - kGPIO_ActiveHighTriggerOutputEnable : Enable active high-trigger output (if the trigger states exit).
  - kGPIO_ActiveLowTriggerOutputEnable : Enable active low-trigger output (if the trigger states exit).

static inline void GPIO_SetPinInterruptChannel(GPIO_Type *base, uint32_t pin, *gpio_interrupt_selection_t* selection)

Configures the gpio pin interrupt/DMA request/trigger output channel selection.

Parameters

- base – GPIO peripheral base pointer.
- pin – GPIO pin number.
- selection – GPIO pin interrupt output selection.
  - kGPIO_InterruptOutput0: Interrupt/DMA request/trigger output 0.
  - kGPIO_InterruptOutput1 : Interrupt/DMA request/trigger output 1.

uint32_t GPIO_GpioGetInterruptFlags(GPIO_Type *base)

Read the GPIO interrupt status flags.

Parameters

- base – GPIO peripheral base pointer. (GPIOA, GPIOB, GPIOC, and so on.)

Returns
The current GPIO's interrupt status flag. '1' means the related pin's flag is set, '0' means the related pin's flag not set. For example, the return value 0x00010001 means the pin 0 and 17 have the interrupt pending.

uint32_t GPIO_GpioGetInterruptChannelFlags(GPIO_Type *base, uint32_t channel)

Read the GPIO interrupt status flags based on selected interrupt channel(IRQS).

Parameters

- base – GPIO peripheral base pointer. (GPIOA, GPIOB, GPIOC, and so on.)
- channel – '0' means selete interrupt channel 0, '1' means selete interrupt channel 1.

Returns
The current GPIO's interrupt status flag based on the selected interrupt channel. '1' means the related pin's flag is set, '0' means the related pin's flag not set. For example, the return value 0x00010001 means the pin 0 and 17 have the interrupt pending.

uint8_t GPIO_PinGetInterruptFlag(GPIO_Type *base, uint32_t pin)

Read individual pin's interrupt status flag.

Parameters

- base – GPIO peripheral base pointer. (GPIOA, GPIOB, GPIOC, and so on)
- pin – GPIO specific pin number.

Returns
The current selected pin's interrupt status flag.

void GPIO_GpioClearInterruptFlags(GPIO_Type *base, uint32_t mask)

Clears GPIO pin interrupt status flags.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

void GPIO_GpioClearInterruptChannelFlags(GPIO_Type *base, uint32_t mask, uint32_t channel)

Clears GPIO pin interrupt status flags based on selected interrupt channel(IRQS).

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro
- channel – '0' means selete interrupt channel 0, '1' means selete interrupt channel 1.

void GPIO_PinClearInterruptFlag(GPIO_Type *base, uint32_t pin)

Clear GPIO individual pin's interrupt status flag.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on).
- pin – GPIO specific pin number.

static inline void GPIO_SetMultipleInterruptPinsConfig(GPIO_Type *base, uint32_t mask, *gpio_interrupt_config_t* config)

Sets the GPIO interrupt configuration in PCR register for multiple pins.

Parameters

- base – GPIO peripheral base pointer.
- mask – GPIO pin number macro.
- config – GPIO pin interrupt configuration.
  - kGPIO_InterruptStatusFlagDisabled: Interrupt disabled.
  - kGPIO_DMARisingEdge : DMA request on rising edge(if the DMA requests exit).
  - kGPIO_DMAFallingEdge: DMA request on falling edge(if the DMA requests exit).
  - kGPIO_DMAEitherEdge : DMA request on either edge(if the DMA requests exit).
  - kGPIO_FlagRisingEdge : Flag sets on rising edge(if the Flag states exit).
  - kGPIO_FlagFallingEdge : Flag sets on falling edge(if the Flag states exit).
  - kGPIO_FlagEitherEdge : Flag sets on either edge(if the Flag states exit).
  - kGPIO_InterruptLogicZero : Interrupt when logic zero.
  - kGPIO_InterruptRisingEdge : Interrupt on rising edge.
  - kGPIO_InterruptFallingEdge: Interrupt on falling edge.
  - kGPIO_InterruptEitherEdge : Interrupt on either edge.
  - kGPIO_InterruptLogicOne : Interrupt when logic one.
  - kGPIO_ActiveHighTriggerOutputEnable : Enable active high-trigger output (if the trigger states exit).
  - kGPIO_ActiveLowTriggerOutputEnable : Enable active low-trigger output (if the trigger states exit)..

void GPIO_CheckAttributeBytes(GPIO_Type *base, *gpio_checker_attribute_t* attribute)

brief The GPIO module supports a device-specific number of data ports, organized as 32-bit words/8-bit Bytes. Each 32-bit/8-bit data port includes a GACR register, which defines the byte-level attributes required for a successful access to the GPIO programming model. If the GPIO module's GACR register organized as 32-bit words, the attribute controls for the 4 data bytes in the GACR follow a standard little endian data convention.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- attribute – GPIO checker attribute

2.24 I3C: I3C Driver

FSL_I3C_DRIVER_VERSION

I3C driver version.

I3C status return codes.

Values:

enumerator kStatus_I3C_Busy

The master is already performing a transfer.

enumerator kStatus_I3C_Idle
The slave driver is idle.

enumerator kStatus_I3C_Nak
The slave device sent a NAK in response to an address.

enumerator kStatus_I3C_WriteAbort
The slave device sent a NAK in response to a write.

enumerator kStatus_I3C_Term
The master terminates slave read.

enumerator kStatus_I3C_HdrParityError
Parity error from DDR read.

enumerator kStatus_I3C_CrcError
CRC error from DDR read.

enumerator kStatus_I3C_ReadFifoError
Read from M/SRDATAB register when FIFO empty.

enumerator kStatus_I3C_WriteFifoError
Write to M/SWDATAB register when FIFO full.

enumerator kStatus_I3C_MsgError
Message SDR/DDR mismatch or read/write message in wrong state

enumerator kStatus_I3C_InvalidReq
Invalid use of request.

enumerator kStatus_I3C_Timeout
The module has stalled too long in a frame.

enumerator kStatus_I3C_SlaveCountExceed
The I3C slave count has exceed the definition in I3C_MAX_DEVCNT.

enumerator kStatus_I3C_IBIWon
The I3C slave event IBI or MR or HJ won the arbitration on a header address.

enumerator kStatus_I3C_OverrunError
Slave internal from-bus buffer/FIFO overrun.

enumerator kStatus_I3C_UnderrunError
Slave internal to-bus buffer/FIFO underrun

enumerator kStatus_I3C_UnderrunNak
Slave internal from-bus buffer/FIFO underrun and NACK error

enumerator kStatus_I3C_InvalidStart
Slave invalid start flag

enumerator kStatus_I3C_SdrParityError
SDR parity error

enumerator kStatus_I3C_S0S1Error
S0 or S1 error

enum _i3c_hdr_mode
I3C HDR modes.

Values:

enumerator kI3C_HDRModeNone

enumerator kI3C_HDRModeDDR

enumerator kI3C_HDRModeTSP

enumerator kI3C_HDRModeTSL

typedef enum *_i3c_hdr_mode* i3c_hdr_mode_t

I3C HDR modes.

typedef struct *_i3c_device_info* i3c_device_info_t

I3C device information.

I3C_RETRY_TIMES

Max loops to wait for I3C operation status complete.

This is the maximum number of loops to wait for I3C operation status complete. If set to 0, it will wait indefinitely.

I3C_MAX_DEVCNT

I3C_IBI_BUFF_SIZE

struct _i3c_device_info

#include <fsl_i3c.h> I3C device information.

Public Members

uint8_t dynamicAddr

Device dynamic address.

uint8_t staticAddr

Static address.

uint8_t dcr

Device characteristics register information.

uint8_t bcr

Bus characteristics register information.

uint16_t vendorID

Device vendor ID(manufacture ID).

uint32_t partNumber

Device part number info

uint16_t maxReadLength

Maximum read length.

uint16_t maxWriteLength

Maximum write length.

uint8_t hdrMode

Support hdr mode, could be OR logic in i3c_hdr_mode.

2.25 I3C Common Driver

typedef struct *_i3c_config* i3c_config_t

Structure with settings to initialize the I3C module, could both initialize master and slave functionality.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_GetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

uint32_t I3C_GetInstance(I3C_Type *base)

Get which instance current I3C is used.

Parameters

- base – The I3C peripheral base address.

void I3C_GetDefaultConfig(*i3c_config_t* *config)

Provides a default configuration for the I3C peripheral, the configuration covers both master functionality and slave functionality.

This function provides the following default configuration for I3C:

```
config->enableMaster                = kI3C_MasterCapable;
config->disableTimeout              = false;
config->hKeep                       = kI3C_MasterHighKeeperNone;
config->enableOpenDrainStop         = true;
config->enableOpenDrainHigh         = true;
config->baudRate_Hz.i2cBaud         = 400000U;
config->baudRate_Hz.i3cPushPullBaud = 12500000U;
config->baudRate_Hz.i3cOpenDrainBaud = 2500000U;
config->masterDynamicAddress        = 0x0AU;
config->slowClock_Hz                = 1000000U;
config->enableSlave                 = true;
config->vendorID                    = 0x11BU;
config->enableRandomPart            = false;
config->partNumber                  = 0;
config->dcr                         = 0;
config->bcr = 0;
config->hdrMode                     = (uint8_t)kI3C_HDRModeDDR;
config->nakAllRequest               = false;
config->ignoreS0S1Error             = false;
config->offline                     = false;
config->matchSlaveStartStop = false;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the common I3C driver with I3C_Init().

Parameters

- config – **[out]** User provided configuration structure for default values. Refer to i3c_config_t.

void I3C_Init(I3C_Type *base, const *i3c_config_t* *config, uint32_t sourceClock_Hz)

Initializes the I3C peripheral. This function enables the peripheral clock and initializes the I3C peripheral as described by the user provided configuration. This will initialize both the master peripheral and slave peripheral so that I3C module could work as pure master, pure slave or secondary master, etc. A software reset is performed prior to configuration.

Parameters

- base – The I3C peripheral base address.
- config – User provided peripheral configuration. Use I3C_GetDefaultConfig() to get a set of defaults that you can override.

- sourceClock_Hz – Frequency in Hertz of the I3C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

struct _i3c_config

#include <fsl_i3c.h> Structure with settings to initialize the I3C module, could both initialize master and slave functionality.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_GetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

i3c_master_enable_t enableMaster
Enable master mode.

bool disableTimeout
Whether to disable timeout to prevent the ERRWARN.

i3c_master_hkeep_t hKeep
High keeper mode setting.

bool enableOpenDrainStop
Whether to emit open-drain speed STOP.

bool enableOpenDrainHigh
Enable Open-Drain High to be 1 PPBAUD count for i3c messages, or 1 ODBAUD.

i3c_baudrate_hz_t baudRate_Hz
Desired baud rate settings.

i3c_start_scl_delay_t startSclDelay
I3C SCL delay after START.

i3c_start_scl_delay_t restartSclDelay
I3C SCL delay after Repeated START.

uint8_t masterDynamicAddress
Main master dynamic address configuration.

uint32_t maxWriteLength
Maximum write length.

uint32_t maxReadLength
Maximum read length.

bool enableSlave
Whether to enable slave.

uint8_t staticAddr
Static address.

uint16_t vendorID
Device vendor ID(manufacture ID).

uint32_t partNumber
Device part number info

uint8_t dcr
Device characteristics register information.

uint8_t bcr
Bus characteristics register information.

uint8_t hdrMode
Support hdr mode, could be OR logic in enumeration:i3c_hdr_mode_t.

bool nakAllRequest
Whether to reply NAK to all requests except broadcast CCC.

bool ignoreS0S1Error
Whether to ignore S0/S1 error in SDR mode.

bool offline
Whether to wait 60 us of bus quiet or HDR request to ensure slave track SDR mode safely.

bool matchSlaveStartStop
Whether to assert start/stop status only the time slave is addressed.

2.26 I3C Master Driver

void I3C_MasterGetDefaultConfig(*i3c_master_config_t* *masterConfig)

Provides a default configuration for the I3C master peripheral.

This function provides the following default configuration for the I3C master peripheral:

```
masterConfig->enableMaster          = kI3C_MasterOn;
masterConfig->disableTimeout        = false;
masterConfig->hKeep                 = kI3C_MasterHighKeeperNone;
masterConfig->enableOpenDrainStop   = true;
masterConfig->enableOpenDrainHigh   = true;
masterConfig->baudRate_Hz           = 100000U;
masterConfig->busType               = kI3C_TypeI2C;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with I3C_MasterInit().

Parameters

- masterConfig – **[out]** User provided configuration structure for default values. Refer to i3c_master_config_t.

void I3C_MasterInit(I3C_Type *base, const *i3c_master_config_t* *masterConfig, uint32_t sourceClock_Hz)

Initializes the I3C master peripheral.

This function enables the peripheral clock and initializes the I3C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

- base – The I3C peripheral base address.
- masterConfig – User provided peripheral configuration. Use I3C_MasterGetDefaultConfig() to get a set of defaults that you can override.
- sourceClock_Hz – Frequency in Hertz of the I3C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

void I3C_MasterDeinit(I3C_Type *base)

Deinitializes the I3C master peripheral.

This function disables the I3C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- base – The I3C peripheral base address.

status_t I3C_MasterCheckAndClearError(I3C_Type *base, uint32_t status)

status_t I3C_MasterWaitForCtrlDone(I3C_Type *base, bool waitIdle)

status_t I3C_CheckForBusyBus(I3C_Type *base)

static inline void I3C_MasterEnable(I3C_Type *base, *i3c_master_enable_t* enable)

Set I3C module master mode.

Parameters

- base – The I3C peripheral base address.
- enable – Enable master mode.

void I3C_SlaveGetDefaultConfig(*i3c_slave_config_t* *slaveConfig)

Provides a default configuration for the I3C slave peripheral.

This function provides the following default configuration for the I3C slave peripheral:

```
slaveConfig->enableslave          = true;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the slave driver with I3C_SlaveInit().

Parameters

- slaveConfig – **[out]** User provided configuration structure for default values. Refer to i3c_slave_config_t.

void I3C_SlaveInit(I3C_Type *base, const *i3c_slave_config_t* *slaveConfig, uint32_t slowClock_Hz)

Initializes the I3C slave peripheral.

This function enables the peripheral clock and initializes the I3C slave peripheral as described by the user provided configuration.

Parameters

- base – The I3C peripheral base address.
- slaveConfig – User provided peripheral configuration. Use I3C_SlaveGetDefaultConfig() to get a set of defaults that you can override.
- slowClock_Hz – Frequency in Hertz of the I3C slow clock. Used to calculate the bus match condition values. If FSL_FEATURE_I3C_HAS_NO_SCONFIG_BAMATCH defines as 1, this parameter is useless.

void I3C_SlaveDeinit(I3C_Type *base)

Deinitializes the I3C slave peripheral.

This function disables the I3C slave peripheral and gates the clock.

Parameters

- base – The I3C peripheral base address.

static inline void I3C_SlaveEnable(I3C_Type *base, bool isEnable)

Enable/Disable Slave.

Parameters

- base – The I3C peripheral base address.
- isEnable – Enable or disable.

static inline uint32_t I3C_MasterGetStatusFlags(I3C_Type *base)

Gets the I3C master status flags.

A bit mask with the state of all I3C master status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

_i3c_master_flags

Parameters

- base – The I3C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

static inline void I3C_MasterClearStatusFlags(I3C_Type *base, uint32_t statusMask)

Clears the I3C master status flag state.

The following status register flags can be cleared:

- kI3C_MasterSlaveStartFlag
- kI3C_MasterControlDoneFlag
- kI3C_MasterCompleteFlag
- kI3C_MasterArbitrationWonFlag
- kI3C_MasterSlave2MasterFlag

Attempts to clear other flags has no effect.

See also:

_i3c_master_flags.

Parameters

- base – The I3C peripheral base address.
- statusMask – A bitmask of status flags that are to be cleared. The mask is composed of _i3c_master_flags enumerators OR'd together. You may pass the result of a previous call to I3C_MasterGetStatusFlags().

static inline uint32_t I3C_MasterGetErrorStatusFlags(I3C_Type *base)

Gets the I3C master error status flags.

A bit mask with the state of all I3C master error status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

_i3c_master_error_flags

Parameters

- base – The I3C peripheral base address.

Returns

State of the error status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

static inline void I3C_MasterClearErrorStatusFlags(I3C_Type *base, uint32_t statusMask)

Clears the I3C master error status flag state.

See also:

_i3c_master_error_flags.

Parameters

- base – The I3C peripheral base address.
- statusMask – A bitmask of error status flags that are to be cleared. The mask is composed of _i3c_master_error_flags enumerators OR'd together. You may pass the result of a previous call to I3C_MasterGetStatusFlags().

i3c_master_state_t I3C_MasterGetState(I3C_Type *base)

Gets the I3C master state.

Parameters

- base – The I3C peripheral base address.

Returns

I3C master state.

static inline uint32_t I3C_SlaveGetStatusFlags(I3C_Type *base)

Gets the I3C slave status flags.

A bit mask with the state of all I3C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

_i3c_slave_flags

Parameters

- base – The I3C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

static inline void I3C_SlaveClearStatusFlags(I3C_Type *base, uint32_t statusMask)

Clears the I3C slave status flag state.

The following status register flags can be cleared:

- kI3C_SlaveBusStartFlag

- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag

Attempts to clear other flags has no effect.

See also:

_i3c_slave_flags.

Parameters

- base – The I3C peripheral base address.
- statusMask – A bitmask of status flags that are to be cleared. The mask is composed of _i3c_slave_flags enumerators OR'd together. You may pass the result of a previous call to I3C_SlaveGetStatusFlags().

static inline uint32_t I3C_SlaveGetErrorStatusFlags(I3C_Type *base)

Gets the I3C slave error status flags.

A bit mask with the state of all I3C slave error status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

_i3c_slave_error_flags

Parameters

- base – The I3C peripheral base address.

Returns

State of the error status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

static inline void I3C_SlaveClearErrorStatusFlags(I3C_Type *base, uint32_t statusMask)

Clears the I3C slave error status flag state.

See also:

_i3c_slave_error_flags.

Parameters

- base – The I3C peripheral base address.
- statusMask – A bitmask of error status flags that are to be cleared. The mask is composed of _i3c_slave_error_flags enumerators OR'd together. You may pass the result of a previous call to I3C_SlaveGetErrorStatusFlags().

i3c_slave_activity_state_t I3C_SlaveGetActivityState(I3C_Type *base)

Gets the I3C slave state.

Parameters

- base – The I3C peripheral base address.

Returns

I3C slave activity state, refer i3c_slave_activity_state_t.

status_t I3C_SlaveCheckAndClearError(I3C_Type *base, uint32_t status)

static inline void I3C_MasterEnableInterrupts(I3C_Type *base, uint32_t interruptMask)

Enables the I3C master interrupt requests.

All flags except kI3C_MasterBetweenFlag and kI3C_MasterNackDetectFlag can be enabled as interrupts.

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See _i3c_master_flags for the set of constants that should be OR'd together to form the bit mask.

static inline void I3C_MasterDisableInterrupts(I3C_Type *base, uint32_t interruptMask)

Disables the I3C master interrupt requests.

All flags except kI3C_MasterBetweenFlag and kI3C_MasterNackDetectFlag can be enabled as interrupts.

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See _i3c_master_flags for the set of constants that should be OR'd together to form the bit mask.

static inline uint32_t I3C_MasterGetEnabledInterrupts(I3C_Type *base)

Returns the set of currently enabled I3C master interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of _i3c_master_flags enumerators OR'd together to indicate the set of enabled interrupts.

static inline uint32_t I3C_MasterGetPendingInterrupts(I3C_Type *base)

Returns the set of pending I3C master interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of _i3c_master_flags enumerators OR'd together to indicate the set of pending interrupts.

static inline void I3C_SlaveEnableInterrupts(I3C_Type *base, uint32_t interruptMask)

Enables the I3C slave interrupt requests.

Only below flags can be enabled as interrupts.

- kI3C_SlaveBusStartFlag
- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag
- kI3C_SlaveRxReadyFlag
- kI3C_SlaveTxReadyFlag
- kI3C_SlaveDynamicAddrChangedFlag
- kI3C_SlaveReceivedCCCFlag
- kI3C_SlaveErrorFlag

- kI3C_SlaveHDRCommandMatchFlag
- kI3C_SlaveCCCHandledFlag
- kI3C_SlaveEventSentFlag

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See _i3c_slave_flags for the set of constants that should be OR'd together to form the bit mask.

static inline void I3C_SlaveDisableInterrupts(I3C_Type *base, uint32_t interruptMask)

Disables the I3C slave interrupt requests.

Only below flags can be disabled as interrupts.

- kI3C_SlaveBusStartFlag
- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag
- kI3C_SlaveRxReadyFlag
- kI3C_SlaveTxReadyFlag
- kI3C_SlaveDynamicAddrChangedFlag
- kI3C_SlaveReceivedCCCFlag
- kI3C_SlaveErrorFlag
- kI3C_SlaveHDRCommandMatchFlag
- kI3C_SlaveCCCHandledFlag
- kI3C_SlaveEventSentFlag

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See _i3c_slave_flags for the set of constants that should be OR'd together to form the bit mask.

static inline uint32_t I3C_SlaveGetEnabledInterrupts(I3C_Type *base)

Returns the set of currently enabled I3C slave interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of _i3c_slave_flags enumerators OR'd together to indicate the set of enabled interrupts.

static inline uint32_t I3C_SlaveGetPendingInterrupts(I3C_Type *base)

Returns the set of pending I3C slave interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of _i3c_slave_flags enumerators OR'd together to indicate the set of pending interrupts.

static inline void I3C_MasterEnableDMA(I3C_Type *base, bool enableTx, bool enableRx, uint32_t width)

Enables or disables I3C master DMA requests.

Parameters

- base – The I3C peripheral base address.
- enableTx – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- enableRx – Enable flag for receive DMA request. Pass true for enable, false for disable.
- width – DMA read/write unit in bytes.

static inline uint32_t I3C_MasterGetTxFifoAddress(I3C_Type *base, uint32_t width)

Gets I3C master transmit data register address for DMA transfer.

Parameters

- base – The I3C peripheral base address.
- width – DMA read/write unit in bytes.

Returns

The I3C Master Transmit Data Register address.

static inline uint32_t I3C_MasterGetRxFifoAddress(I3C_Type *base, uint32_t width)

Gets I3C master receive data register address for DMA transfer.

Parameters

- base – The I3C peripheral base address.
- width – DMA read/write unit in bytes.

Returns

The I3C Master Receive Data Register address.

static inline void I3C_SlaveEnableDMA(I3C_Type *base, bool enableTx, bool enableRx, uint32_t width)

Enables or disables I3C slave DMA requests.

Parameters

- base – The I3C peripheral base address.
- enableTx – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- enableRx – Enable flag for receive DMA request. Pass true for enable, false for disable.
- width – DMA read/write unit in bytes.

static inline uint32_t I3C_SlaveGetTxFifoAddress(I3C_Type *base, uint32_t width)

Gets I3C slave transmit data register address for DMA transfer.

Parameters

- base – The I3C peripheral base address.
- width – DMA read/write unit in bytes.

Returns

The I3C Slave Transmit Data Register address.

static inline uint32_t I3C_SlaveGetRxFifoAddress(I3C_Type *base, uint32_t width)

Gets I3C slave receive data register address for DMA transfer.

Parameters

- base – The I3C peripheral base address.
- width – DMA read/write unit in bytes.

Returns

The I3C Slave Receive Data Register address.

static inline void I3C_MasterSetWatermarks(I3C_Type *base, *i3c_tx_trigger_level_t* txLvl, *i3c_rx_trigger_level_t* rxLvl, bool flushTx, bool flushRx)

Sets the watermarks for I3C master FIFOs.

Parameters

- base – The I3C peripheral base address.
- txLvl – Transmit FIFO watermark level. The kI3C_MasterTxReadyFlag flag is set whenever the number of words in the transmit FIFO reaches *txLvl*.
- rxLvl – Receive FIFO watermark level. The kI3C_MasterRxReadyFlag flag is set whenever the number of words in the receive FIFO reaches *rxLvl*.
- flushTx – true if TX FIFO is to be cleared, otherwise TX FIFO remains unchanged.
- flushRx – true if RX FIFO is to be cleared, otherwise RX FIFO remains unchanged.

static inline void I3C_MasterGetFifoCounts(I3C_Type *base, size_t *rxCount, size_t *txCount)

Gets the current number of bytes in the I3C master FIFOs.

Parameters

- base – The I3C peripheral base address.
- txCount – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.
- rxCount – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

static inline void I3C_SlaveSetWatermarks(I3C_Type *base, *i3c_tx_trigger_level_t* txLvl, *i3c_rx_trigger_level_t* rxLvl, bool flushTx, bool flushRx)

Sets the watermarks for I3C slave FIFOs.

Parameters

- base – The I3C peripheral base address.
- txLvl – Transmit FIFO watermark level. The kI3C_SlaveTxReadyFlag flag is set whenever the number of words in the transmit FIFO reaches *txLvl*.
- rxLvl – Receive FIFO watermark level. The kI3C_SlaveRxReadyFlag flag is set whenever the number of words in the receive FIFO reaches *rxLvl*.
- flushTx – true if TX FIFO is to be cleared, otherwise TX FIFO remains unchanged.
- flushRx – true if RX FIFO is to be cleared, otherwise RX FIFO remains unchanged.

static inline void I3C_SlaveGetFifoCounts(I3C_Type *base, size_t *rxCount, size_t *txCount)

Gets the current number of bytes in the I3C slave FIFOs.

Parameters

- base – The I3C peripheral base address.
- txCount – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.
- rxCount – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

void I3C_MasterSetBaudRate(I3C_Type *base, const *i3c_baudrate_hz_t* *baudRate_Hz, uint32_t sourceClock_Hz)

Sets the I3C bus frequency for master transactions.

The I3C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

Parameters

- base – The I3C peripheral base address.
- baudRate_Hz – Pointer to structure of requested bus frequency in Hertz.
- sourceClock_Hz – I3C functional clock frequency in Hertz.

static inline bool I3C_MasterGetBusIdleState(I3C_Type *base)

Returns whether the bus is idle.

Requires the master mode to be enabled.

Parameters

- base – The I3C peripheral base address.

Return values

- true – Bus is busy.
- false – Bus is idle.

status_t I3C_MasterStartWithRxSize(I3C_Type *base, *i3c_bus_type_t* type, uint8_t address, *i3c_direction_t* dir, uint8_t rxSize)

Sends a START signal and slave address on the I2C/I3C bus, receive size is also specified in the call.

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the a address parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

- base – The I3C peripheral base address.
- type – The bus type to use in this transaction.
- address – 7-bit slave device address, in bits [6:0].
- dir – Master transfer direction, either kI3C_Read or kI3C_Write. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.
- rxSize – Read terminate size for the followed read transfer, limit to 255 bytes.

Return values

- kStatus_Success – START signal and address were successfully enqueued in the transmit FIFO.
- kStatus_I3C_Busy – Another master is currently utilizing the bus.

status_t I3C_MasterStart(I3C_Type *base, *i3c_bus_type_t* type, uint8_t address, *i3c_direction_t* dir)

Sends a START signal and slave address on the I2C/I3C bus.

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the *address* parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

- base – The I3C peripheral base address.
- type – The bus type to use in this transaction.
- address – 7-bit slave device address, in bits [6:0].
- dir – Master transfer direction, either kI3C_Read or kI3C_Write. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

- kStatus_Success – START signal and address were successfully enqueued in the transmit FIFO.
- kStatus_I3C_Busy – Another master is currently utilizing the bus.

status_t I3C_MasterRepeatedStartWithRxSize(I3C_Type *base, *i3c_bus_type_t* type, uint8_t address, *i3c_direction_t* dir, uint8_t rxSize)

Sends a repeated START signal and slave address on the I2C/I3C bus, receive size is also specified in the call.

This function is used to send a Repeated START signal when a transfer is already in progress. Like I3C_MasterStart(), it also sends the specified 7-bit address. Call this API also configures the read terminate size for the following read transfer. For example, set the rxSize = 2, the following read transfer will be terminated after two bytes of data received. Write transfer will not be affected by the rxSize configuration.

Note: This function exists primarily to maintain compatible APIs between I3C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

- base – The I3C peripheral base address.
- type – The bus type to use in this transaction.
- address – 7-bit slave device address, in bits [6:0].
- dir – Master transfer direction, either kI3C_Read or kI3C_Write. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.
- rxSize – Read terminate size for the followed read transfer, limit to 255 bytes.

Return values

kStatus_Success – Repeated START signal and address were successfully enqueued in the transmit FIFO.

static inline *status_t* I3C_MasterRepeatedStart(I3C_Type *base, *i3c_bus_type_t* type, uint8_t address, *i3c_direction_t* dir)

Sends a repeated START signal and slave address on the I2C/I3C bus.

This function is used to send a Repeated START signal when a transfer is already in progress. Like I3C_MasterStart(), it also sends the specified 7-bit address.

Note: This function exists primarily to maintain compatible APIs between I3C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

- base – The I3C peripheral base address.
- type – The bus type to use in this transaction.
- address – 7-bit slave device address, in bits [6:0].
- dir – Master transfer direction, either kI3C_Read or kI3C_Write. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

kStatus_Success – Repeated START signal and address were successfully enqueued in the transmit FIFO.

status_t I3C_MasterSend(I3C_Type *base, const void *txBuff, size_t txSize, uint32_t flags)

Performs a polling send transfer on the I2C/I3C bus.

Sends up to *txSize* number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns kStatus_I3C_Nak.

Parameters

- base – The I3C peripheral base address.
- txBuff – The pointer to the data to be transferred.
- txSize – The length in bytes of the data to be transferred.
- flags – Bit mask of options for the transfer. See enumeration _i3c_master_transfer_flags for available options.

Return values

- kStatus_Success – Data was sent successfully.
- kStatus_I3C_Busy – Another master is currently utilizing the bus.
- kStatus_I3C_Timeout – The module has stalled too long in a frame.
- kStatus_I3C_Nak – The slave device sent a NAK in response to an address.
- kStatus_I3C_WriteAbort – The slave device sent a NAK in response to a write.
- kStatus_I3C_MsgError – Message SDR/DDR mismatch or read/write message in wrong state.
- kStatus_I3C_WriteFifoError – Write to M/SWDATAB register when FIFO full.
- kStatus_I3C_InvalidReq – Invalid use of request.

status_t I3C_MasterReceive(I3C_Type *base, void *rxBuff, size_t rxSize, uint32_t flags)

Performs a polling receive transfer on the I2C/I3C bus.

Parameters

- base – The I3C peripheral base address.
- rxBuff – The pointer to the data to be transferred.
- rxSize – The length in bytes of the data to be transferred.
- flags – Bit mask of options for the transfer. See enumeration _i3c_master_transfer_flags for available options.

Return values

- kStatus_Success – Data was received successfully.
- kStatus_I3C_Busy – Another master is currently utilizing the bus.
- kStatus_I3C_Timeout – The module has stalled too long in a frame.
- kStatus_I3C_Term – The master terminates slave read.
- kStatus_I3C_HdrParityError – Parity error from DDR read.
- kStatus_I3C_CrcError – CRC error from DDR read.
- kStatus_I3C_MsgError – Message SDR/DDR mismatch or read/write message in wrong state.
- kStatus_I3C_ReadFifoError – Read from M/SRDATAB register when FIFO empty.
- kStatus_I3C_InvalidReq – Invalid use of request.

status_t I3C_MasterStop(I3C_Type *base)

Sends a STOP signal on the I2C/I3C bus.

This function does not return until the STOP signal is seen on the bus, or an error occurs.

Parameters

- base – The I3C peripheral base address.

Return values

- kStatus_Success – The STOP signal was successfully sent on the bus and the transaction terminated.
- kStatus_I3C_Busy – Another master is currently utilizing the bus.
- kStatus_I3C_Timeout – The module has stalled too long in a frame.
- kStatus_I3C_InvalidReq – Invalid use of request.

void I3C_MasterEmitRequest(I3C_Type *base, *i3c_bus_request_t* masterReq)

I3C master emit request.

Parameters

- base – The I3C peripheral base address.
- masterReq – I3C master request of type i3c_bus_request_t

static inline void I3C_MasterEmitIBIResponse(I3C_Type *base, *i3c_ibi_response_t* ibiResponse)

I3C master emit request.

Parameters

- base – The I3C peripheral base address.
- ibiResponse – I3C master emit IBI response of type i3c_ibi_response_t

void I3C_MasterRegisterIBI(I3C_Type *base, *i3c_register_ibi_addr_t* *ibiRule)

I3C master register IBI rule.

Parameters

- base – The I3C peripheral base address.
- ibiRule – Pointer to ibi rule description of type i3c_register_ibi_addr_t

void I3C_MasterGetIBIRules(I3C_Type *base, *i3c_register_ibi_addr_t* *ibiRule)

I3C master get IBI rule.

Parameters

- base – The I3C peripheral base address.
- ibiRule – Pointer to store the read out ibi rule description.

i3c_ibi_type_t I3C_GetIBIType(I3C_Type *base)

I3C master get IBI Type.

Parameters

- base – The I3C peripheral base address.

Return values

i3c_ibi_type_t – Type of i3c_ibi_type_t.

static inline uint8_t I3C_GetIBIAddress(I3C_Type *base)

I3C master get IBI Address.

Parameters

- base – The I3C peripheral base address.

Return values

The – 8-bit IBI address.

status_t I3C_MasterProcessDAASpecifiedBaudrate(I3C_Type *base, uint8_t *addressList, uint32_t count, *i3c_master_daa_baudrate_t* *daaBaudRate)

Performs a DAA in the i3c bus with specified temporary baud rate.

Parameters

- base – The I3C peripheral base address.
- addressList – The pointer for address list which is used to do DAA.
- count – The address count in the address list.
- daaBaudRate – The temporary baud rate in DAA process, NULL for using initial setting. The initial setting is set back between the completion of the DAA and the return of this function.

Return values

- kStatus_Success – The transaction was started successfully.
- kStatus_I3C_Busy – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.
- kStatus_I3C_SlaveCountExceed – The I3C slave count has exceed the definition in I3C_MAX_DEVCNT.

static inline *status_t* I3C_MasterProcessDAA(I3C_Type *base, uint8_t *addressList, uint32_t count)

Performs a DAA in the i3c bus.

Parameters

- base – The I3C peripheral base address.
- addressList – The pointer for address list which is used to do DAA.
- count – The address count in the address list. The initial setting is set back between the completion of the DAA and the return of this function.

Return values

- kStatus_Success – The transaction was started successfully.
- kStatus_I3C_Busy – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.
- kStatus_I3C_SlaveCountExceed – The I3C slave count has exceed the definition in I3C_MAX_DEVCNT.

i3c_device_info_t *I3C_MasterGetDeviceListAfterDAA(I3C_Type *base, uint8_t *count)

Get device information list after DAA process is done.

Parameters

- base – The I3C peripheral base address.
- count – **[out]** The pointer to store the available device count.

Returns

Pointer to the i3c_device_info_t array.

void I3C_MasterClearDeviceCount(I3C_Type *base)

Clear the global device count which represents current devices number on the bus. When user resets all dynamic addresses on the bus, should call this API.

Parameters

- base – The I3C peripheral base address.

status_t I3C_MasterTransferBlocking(I3C_Type *base, *i3c_master_transfer_t* *transfer)

Performs a master polling transfer on the I2C/I3C bus.

Note: The API does not return until the transfer succeeds or fails due to error happens during transfer.

Parameters

- base – The I3C peripheral base address.
- transfer – Pointer to the transfer structure.

Return values

- kStatus_Success – Data was received successfully.
- kStatus_I3C_Busy – Another master is currently utilizing the bus.
- kStatus_I3C_IBIWon – The I3C slave event IBI or MR or HJ won the arbitration on a header address.
- kStatus_I3C_Timeout – The module has stalled too long in a frame.
- kStatus_I3C_Nak – The slave device sent a NAK in response to an address.
- kStatus_I3C_WriteAbort – The slave device sent a NAK in response to a write.
- kStatus_I3C_Term – The master terminates slave read.
- kStatus_I3C_HdrParityError – Parity error from DDR read.

- kStatus_I3C_CrcError – CRC error from DDR read.
- kStatus_I3C_MsgError – Message SDR/DDR mismatch or read/write message in wrong state.
- kStatus_I3C_ReadFifoError – Read from M/SRDATAB register when FIFO empty.
- kStatus_I3C_WriteFifoError – Write to M/SWDATAB register when FIFO full.
- kStatus_I3C_InvalidReq – Invalid use of request.

status_t I3C_SlaveSend(I3C_Type *base, const void *txBuff, size_t txSize)

Performs a polling send transfer on the I3C bus.

Parameters

- base – The I3C peripheral base address.
- txBuff – The pointer to the data to be transferred.
- txSize – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

status_t I3C_SlaveReceive(I3C_Type *base, void *rxBuff, size_t rxSize)

Performs a polling receive transfer on the I3C bus.

Parameters

- base – The I3C peripheral base address.
- rxBuff – The pointer to the data to be transferred.
- rxSize – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

void I3C_MasterTransferCreateHandle(I3C_Type *base, *i3c_master_handle_t* *handle, const *i3c_master_transfer_callback_t* *callback, void *userData)

Creates a new handle for the I3C master non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the I3C_MasterTransferAbort() API shall be called.

Note: The function also enables the NVIC IRQ for the input I3C. Need to notice that on some SoCs the I3C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- base – The I3C peripheral base address.
- handle – **[out]** Pointer to the I3C master driver handle.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

status_t I3C_MasterTransferNonBlocking(I3C_Type *base, *i3c_master_handle_t* *handle, *i3c_master_transfer_t* *transfer)

Performs a non-blocking transaction on the I2C/I3C bus.

Parameters

- base – The I3C peripheral base address.
- handle – Pointer to the I3C master driver handle.
- transfer – The pointer to the transfer descriptor.

Return values

- kStatus_Success – The transaction was started successfully.
- kStatus_I3C_Busy – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

status_t I3C_MasterTransferGetCount(I3C_Type *base, *i3c_master_handle_t* *handle, size_t *count)

Returns number of bytes transferred so far.

Parameters

- base – The I3C peripheral base address.
- handle – Pointer to the I3C master driver handle.
- count – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

void I3C_MasterTransferAbort(I3C_Type *base, *i3c_master_handle_t* *handle)

Terminates a non-blocking I3C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the I3C peripheral's IRQ priority.

Parameters

- base – The I3C peripheral base address.
- handle – Pointer to the I3C master driver handle.

void I3C_MasterTransferHandleIRQ(I3C_Type *base, void *intHandle)

Reusable routine to handle master interrupts.

Note: This function does not need to be called unless you are reimplementing the non-blocking API's interrupt handler routines to add special functionality.

Parameters

- base – The I3C peripheral base address.
- intHandle – Pointer to the I3C master driver handle.

enum _i3c_master_flags

I3C master peripheral flags.

The following status register flags can be cleared:

- kI3C_MasterSlaveStartFlag

- kI3C_MasterControlDoneFlag
- kI3C_MasterCompleteFlag
- kI3C_MasterArbitrationWonFlag
- kI3C_MasterSlave2MasterFlag

All flags except kI3C_MasterBetweenFlag and kI3C_MasterNackDetectFlag can be enabled as interrupts.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI3C_MasterBetweenFlag
Between messages/DAAs flag

enumerator kI3C_MasterNackDetectFlag
NACK detected flag

enumerator kI3C_MasterSlaveStartFlag
Slave request start flag

enumerator kI3C_MasterControlDoneFlag
Master request complete flag

enumerator kI3C_MasterCompleteFlag
Transfer complete flag

enumerator kI3C_MasterRxReadyFlag
Rx data ready in Rx buffer flag

enumerator kI3C_MasterTxReadyFlag
Tx buffer ready for Tx data flag

enumerator kI3C_MasterArbitrationWonFlag
Header address won arbitration flag

enumerator kI3C_MasterErrorFlag
Error occurred flag

enumerator kI3C_MasterSlave2MasterFlag
Switch from slave to master flag

enumerator kI3C_MasterClearFlags

enum _i3c_master_error_flags
I3C master error flags to indicate the causes.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI3C_MasterErrorNackFlag
Slave NACKed the last address

enumerator kI3C_MasterErrorWriteAbortFlag
Slave NACKed the write data

enumerator kI3C_MasterErrorParityFlag
Parity error from DDR read

enumerator kI3C_MasterErrorCrcFlag
CRC error from DDR read

enumerator kI3C_MasterErrorReadFlag
Read from MRDATAB register when FIFO empty

enumerator kI3C_MasterErrorWriteFlag
Write to MWDATAB register when FIFO full

enumerator kI3C_MasterErrorMsgFlag
Message SDR/DDR mismatch or read/write message in wrong state

enumerator kI3C_MasterErrorInvalidReqFlag
Invalid use of request

enumerator kI3C_MasterErrorTimeoutFlag
The module has stalled too long in a frame

enumerator kI3C_MasterAllErrorFlags
All error flags

enum _i3c_master_state
I3C working master state.

Values:

enumerator kI3C_MasterStateIdle
Bus stopped.

enumerator kI3C_MasterStateSlvReq
Bus stopped but slave holding SDA low.

enumerator kI3C_MasterStateMsgSdr
In SDR Message mode from using MWMSG_SDR.

enumerator kI3C_MasterStateNormAct
In normal active SDR mode.

enumerator kI3C_MasterStateDdr
In DDR Message mode.

enumerator kI3C_MasterStateDaa
In ENTDAA mode.

enumerator kI3C_MasterStateIbiAck
Waiting on IBI ACK/NACK decision.

enumerator kI3C_MasterStateIbiRcv
Receiving IBI.

enum _i3c_master_enable
I3C master enable configuration.

Values:

enumerator kI3C_MasterOff
Master off.

enumerator kI3C_MasterOn
Master on.

enumerator kI3C_MasterCapable
Master capable.

enum _i3c_master_hkeep

I3C high keeper configuration.

Values:

enumerator kI3C_MasterHighKeeperNone

Use PUR to hold SCL high.

enumerator kI3C_MasterHighKeeperWiredIn

Use pin_HK controls.

enumerator kI3C_MasterPassiveSDA

Hi-Z for Bus Free and hold SDA.

enumerator kI3C_MasterPassiveSDASCL

Hi-Z both for Bus Free, and can Hi-Z SDA for hold.

enum _i3c_bus_request

Emits the requested operation when doing in pieces vs. by message.

Values:

enumerator kI3C_RequestNone

No request.

enumerator kI3C_RequestEmitStartAddr

Request to emit start and address on bus.

enumerator kI3C_RequestEmitStop

Request to emit stop on bus.

enumerator kI3C_RequestIbiAckNack

Manual IBI ACK or NACK.

enumerator kI3C_RequestProcessDAA

Process DAA.

enumerator kI3C_RequestForceExit

Request to force exit.

enumerator kI3C_RequestAutoIbi

Hold in stopped state, but Auto-emit START,7E.

enum _i3c_bus_type

Bus type with EmitStartAddr.

Values:

enumerator kI3C_TypeI3CSdr

SDR mode of I3C.

enumerator kI3C_TypeI2C

Standard i2c protocol.

enumerator kI3C_TypeI3CDdr

HDR-DDR mode of I3C.

enum _i3c_ibi_response

IBI response.

Values:

enumerator kI3C_IbiRespAck

ACK with no mandatory byte.

enumerator kI3C_IbiRespNack
NACK.

enumerator kI3C_IbiRespAckMandatory
ACK with mandatory byte.

enumerator kI3C_IbiRespManual
Reserved.

enum _i3c_ibi_type
IBI type.

Values:

enumerator kI3C_IbiNormal
In-band interrupt.

enumerator kI3C_IbiHotJoin
slave hot join.

enumerator kI3C_IbiMasterRequest
slave master ship request.

enum _i3c_ibi_state
IBI state.

Values:

enumerator kI3C_IbiReady
In-band interrupt ready state, ready for user to handle.

enumerator kI3C_IbiDataBuffNeed
In-band interrupt need data buffer for data receive.

enumerator kI3C_IbiAckNackPending
In-band interrupt Ack/Nack pending for decision.

enum _i3c_direction
Direction of master and slave transfers.

Values:

enumerator kI3C_Write
Master transmit.

enumerator kI3C_Read
Master receive.

enum _i3c_tx_trigger_level
Watermark of TX int/dma trigger level.

Values:

enumerator kI3C_TxTriggerOnEmpty
Trigger on empty.

enumerator kI3C_TxTriggerUntilOneQuarterOrLess
Trigger on 1/4 full or less.

enumerator kI3C_TxTriggerUntilOneHalfOrLess
Trigger on 1/2 full or less.

enumerator kI3C_TxTriggerUntilOneLessThanFull
Trigger on 1 less than full or less.

enum _i3c_rx_trigger_level

Watermark of RX int/dma trigger level.

Values:

enumerator kI3C_RxTriggerOnNotEmpty

Trigger on not empty.

enumerator kI3C_RxTriggerUntilOneQuarterOrMore

Trigger on 1/4 full or more.

enumerator kI3C_RxTriggerUntilOneHalfOrMore

Trigger on 1/2 full or more.

enumerator kI3C_RxTriggerUntilThreeQuarterOrMore

Trigger on 3/4 full or more.

enum _i3c_rx_term_ops

I3C master read termination operations.

Values:

enumerator kI3C_RxTermDisable

Master doesn't terminate read, used for CCC transfer.

enumerator kI3C_RxAutoTerm

Master auto terminate read after receiving specified bytes(<=255).

enumerator kI3C_RxTermLastByte

Master terminates read at any time after START, no length limitation.

enum _i3c_start_scl_delay

I3C start SCL delay options.

Values:

enumerator kI3C_NoDelay

No delay.

enumerator kI3C_IncreaseSclHalfPeriod

Increases SCL clock period by 1/2.

enumerator kI3C_IncreaseSclOnePeriod

Increases SCL clock period by 1.

enumerator kI3C_IncreaseSclOneAndHalfPeriod

Increases SCL clock period by 1 1/2

enum _i3c_master_transfer_flags

Transfer option flags.

Note: These enumerations are intended to be OR'd together to form a bit mask of options for the _i3c_master_transfer::flags field.

Values:

enumerator kI3C_TransferDefaultFlag

Transfer starts with a start signal, stops with a stop signal.

enumerator kI3C_TransferNoStartFlag

Don't send a start condition, address, and sub address

enumerator kI3C_TransferRepeatedStartFlag
Send a repeated start condition

enumerator kI3C_TransferNoStopFlag
Don't send a stop condition.

enumerator kI3C_TransferWordsFlag
Transfer in words, else transfer in bytes.

enumerator kI3C_TransferDisableRxTermFlag
Disable Rx termination. Note: It's for I3C CCC transfer.

enumerator kI3C_TransferRxAutoTermFlag
Set Rx auto-termination. Note: It's adaptive based on Rx size(<=255 bytes) except in I3C_MasterReceive.

enumerator kI3C_TransferStartWithBroadcastAddr
Start transfer with 0x7E, then read/write data with device address.

typedef enum *_i3c_master_state* i3c_master_state_t
I3C working master state.

typedef enum *_i3c_master_enable* i3c_master_enable_t
I3C master enable configuration.

typedef enum *_i3c_master_hkeep* i3c_master_hkeep_t
I3C high keeper configuration.

typedef enum *_i3c_bus_request* i3c_bus_request_t
Emits the requested operation when doing in pieces vs. by message.

typedef enum *_i3c_bus_type* i3c_bus_type_t
Bus type with EmitStartAddr.

typedef enum *_i3c_ibi_response* i3c_ibi_response_t
IBI response.

typedef enum *_i3c_ibi_type* i3c_ibi_type_t
IBI type.

typedef enum *_i3c_ibi_state* i3c_ibi_state_t
IBI state.

typedef enum *_i3c_direction* i3c_direction_t
Direction of master and slave transfers.

typedef enum *_i3c_tx_trigger_level* i3c_tx_trigger_level_t
Watermark of TX int/dma trigger level.

typedef enum *_i3c_rx_trigger_level* i3c_rx_trigger_level_t
Watermark of RX int/dma trigger level.

typedef enum *_i3c_rx_term_ops* i3c_rx_term_ops_t
I3C master read termination operations.

typedef enum *_i3c_start_scl_delay* i3c_start_scl_delay_t
I3C start SCL delay options.

typedef struct *_i3c_register_ibi_addr* i3c_register_ibi_addr_t
Structure with setting master IBI rules and slave registry.

typedef struct *_i3c_baudrate* i3c_baudrate_hz_t
Structure with I3C baudrate settings.

typedef struct *_i3c_master_daa_baudrate* i3c_master_daa_baudrate_t

I3C DAA baud rate configuration.

typedef struct *_i3c_master_config* i3c_master_config_t

Structure with settings to initialize the I3C master module.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

typedef struct *_i3c_master_transfer* i3c_master_transfer_t

typedef struct *_i3c_master_handle* i3c_master_handle_t

typedef struct *_i3c_master_transfer_callback* i3c_master_transfer_callback_t

i3c master callback functions.

typedef void (*i3c_master_isr_t)(I3C_Type *base, void *handle)

Typedef for master interrupt handler.

struct _i3c_register_ibi_addr

#include <fsl_i3c.h> Structure with setting master IBI rules and slave registry.

Public Members

uint8_t address[5]

Address array for registry.

bool i3cFastStart

Allow the START header to run as push-pull speed if all dynamic addresses take MSB 0.

bool ibiHasPayload

Whether the address array has mandatory IBI byte.

struct _i3c_baudrate

#include <fsl_i3c.h> Structure with I3C baudrate settings.

Public Members

uint32_t i2cBaud

Desired I2C baud rate in Hertz.

uint32_t i3cPushPullBaud

Desired I3C push-pull baud rate in Hertz.

uint32_t i3cOpenDrainBaud

Desired I3C open-drain baud rate in Hertz.

struct _i3c_master_daa_baudrate

#include <fsl_i3c.h> I3C DAA baud rate configuration.

Public Members

uint32_t sourceClock_Hz

FCLK, function clock in Hertz.

uint32_t i3cPushPullBaud
Desired I3C push-pull baud rate in Hertz.

uint32_t i3cOpenDrainBaud
Desired I3C open-drain baud rate in Hertz.

struct __i3c__master__config
#include <fsl_i3c.h> Structure with settings to initialize the I3C master module.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

i3c_master_enable_t enableMaster
Enable master mode.

bool disableTimeout
Whether to disable timeout to prevent the ERRWARN.

i3c_master_hkeep_t hKeep
High keeper mode setting.

bool enableOpenDrainStop
Whether to emit open-drain speed STOP.

bool enableOpenDrainHigh
Enable Open-Drain High to be 1 PPBAUD count for i3c messages, or 1 ODBAUD.

i3c_baudrate_hz_t baudRate__Hz
Desired baud rate settings.

i3c_start_scl_delay_t startSclDelay
I3C SCL delay after START.

i3c_start_scl_delay_t restartSclDelay
I3C SCL delay after Repeated START.

struct __i3c__master__transfer__callback
#include <fsl_i3c.h> i3c master callback functions.

Public Members

void (*slave2Master)(I3C_Type *base, void *userData)
Transfer complete callback

void (*ibiCallback)(I3C_Type *base, *i3c_master_handle_t* *handle, *i3c_ibi_type_t* ibiType, *i3c_ibi_state_t* ibiState)
IBI event callback

void (*transferComplete)(I3C_Type *base, *i3c_master_handle_t* *handle, *status_t* completionStatus, void *userData)
Transfer complete callback

struct __i3c__master__transfer
#include <fsl_i3c.h> Non-blocking transfer descriptor structure.

This structure is used to pass transaction parameters to the I3C_MasterTransferNonBlocking() API.

Public Members

uint32_t flags
: Bit mask of options for the transfer. See enumeration _i3c_master_transfer_flags for available options. Set to 0 or kI3C_TransferDefaultFlag for normal transfers.

uint8_t slaveAddress
: The 7-bit slave address.

i3c_direction_t direction
: Either kI3C_Read or kI3C_Write.

uint32_t subaddress
: Sub address. Transferred MSB first.

size_t subaddressSize
: Length of sub address to send in bytes. Maximum size is 4 bytes.

void *data
: Pointer to data to transfer.

size_t dataSize
: Number of bytes to transfer.

i3c_bus_type_t busType
: bus type.

i3c_ibi_response_t ibiResponse
: ibi response during transfer.

struct __i3c_master_handle
: *#include <fsl_i3c.h>* Driver handle for master non-blocking APIs.

Note: The contents of this structure are private and subject to change.

Public Members

uint8_t state
: Transfer state machine current state.

uint32_t remainingBytes
: Remaining byte count in current state.

i3c_rx_term_ops_t rxTermOps
: Read termination operation.

i3c_master_transfer_t transfer
: Copy of the current transfer info.

uint8_t ibiAddress
: Slave address which request IBI.

uint8_t *ibiBuff
: Pointer to IBI buffer to keep ibi bytes.

size_t ibiPayloadSize
: IBI payload size.

i3c_ibi_type_t ibiType
: IBI type.

i3c_master_transfer_callback_t callback
Callback functions pointer.

void *userData
Application data passed to callback.

2.27 I3C Slave Driver

void I3C_SlaveGetDefaultConfig(*i3c_slave_config_t* *slaveConfig)

Provides a default configuration for the I3C slave peripheral.

This function provides the following default configuration for the I3C slave peripheral:

```
slaveConfig->enableslave          = true;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the slave driver with I3C_SlaveInit().

Parameters

- slaveConfig – **[out]** User provided configuration structure for default values. Refer to i3c_slave_config_t.

void I3C_SlaveInit(I3C_Type *base, const *i3c_slave_config_t* *slaveConfig, uint32_t slowClock_Hz)

Initializes the I3C slave peripheral.

This function enables the peripheral clock and initializes the I3C slave peripheral as described by the user provided configuration.

Parameters

- base – The I3C peripheral base address.
- slaveConfig – User provided peripheral configuration. Use I3C_SlaveGetDefaultConfig() to get a set of defaults that you can override.
- slowClock_Hz – Frequency in Hertz of the I3C slow clock. Used to calculate the bus match condition values. If FSL_FEATURE_I3C_HAS_NO_SCONFIG_BAMATCH defines as 1, this parameter is useless.

void I3C_SlaveDeinit(I3C_Type *base)

Deinitializes the I3C slave peripheral.

This function disables the I3C slave peripheral and gates the clock.

Parameters

- base – The I3C peripheral base address.

static inline void I3C_SlaveEnable(I3C_Type *base, bool isEnable)

Enable/Disable Slave.

Parameters

- base – The I3C peripheral base address.
- isEnable – Enable or disable.

static inline uint32_t I3C_SlaveGetStatusFlags(I3C_Type *base)

Gets the I3C slave status flags.

A bit mask with the state of all I3C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

_i3c_slave_flags

Parameters

- base – The I3C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

static inline void I3C_SlaveClearStatusFlags(I3C_Type *base, uint32_t statusMask)

Clears the I3C slave status flag state.

The following status register flags can be cleared:

- kI3C_SlaveBusStartFlag
- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag

Attempts to clear other flags has no effect.

See also:

_i3c_slave_flags.

Parameters

- base – The I3C peripheral base address.
- statusMask – A bitmask of status flags that are to be cleared. The mask is composed of _i3c_slave_flags enumerators OR'd together. You may pass the result of a previous call to I3C_SlaveGetStatusFlags().

static inline uint32_t I3C_SlaveGetErrorStatusFlags(I3C_Type *base)

Gets the I3C slave error status flags.

A bit mask with the state of all I3C slave error status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

_i3c_slave_error_flags

Parameters

- base – The I3C peripheral base address.

Returns

State of the error status flags:

- 1: related status flag is set.

- 0: related status flag is not set.

static inline void I3C_SlaveClearErrorStatusFlags(I3C_Type *base, uint32_t statusMask)

Clears the I3C slave error status flag state.

See also:

_i3c_slave_error_flags.

Parameters

- base – The I3C peripheral base address.
- statusMask – A bitmask of error status flags that are to be cleared. The mask is composed of _i3c_slave_error_flags enumerators OR'd together. You may pass the result of a previous call to I3C_SlaveGetErrorStatusFlags().

i3c_slave_activity_state_t I3C_SlaveGetActivityState(I3C_Type *base)

Gets the I3C slave state.

Parameters

- base – The I3C peripheral base address.

Returns

I3C slave activity state, refer i3c_slave_activity_state_t.

status_t I3C_SlaveCheckAndClearError(I3C_Type *base, uint32_t status)

static inline void I3C_SlaveEnableInterrupts(I3C_Type *base, uint32_t interruptMask)

Enables the I3C slave interrupt requests.

Only below flags can be enabled as interrupts.

- kI3C_SlaveBusStartFlag
- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag
- kI3C_SlaveRxReadyFlag
- kI3C_SlaveTxReadyFlag
- kI3C_SlaveDynamicAddrChangedFlag
- kI3C_SlaveReceivedCCCFlag
- kI3C_SlaveErrorFlag
- kI3C_SlaveHDRCommandMatchFlag
- kI3C_SlaveCCCHandledFlag
- kI3C_SlaveEventSentFlag

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See _i3c_slave_flags for the set of constants that should be OR'd together to form the bit mask.

static inline void I3C_SlaveDisableInterrupts(I3C_Type *base, uint32_t interruptMask)

Disables the I3C slave interrupt requests.

Only below flags can be disabled as interrupts.

- kI3C_SlaveBusStartFlag

- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag
- kI3C_SlaveRxReadyFlag
- kI3C_SlaveTxReadyFlag
- kI3C_SlaveDynamicAddrChangedFlag
- kI3C_SlaveReceivedCCCFlag
- kI3C_SlaveErrorFlag
- kI3C_SlaveHDRCommandMatchFlag
- kI3C_SlaveCCCHandledFlag
- kI3C_SlaveEventSentFlag

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See _i3c_slave_flags for the set of constants that should be OR'd together to form the bit mask.

static inline uint32_t I3C_SlaveGetEnabledInterrupts(I3C_Type *base)

Returns the set of currently enabled I3C slave interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of _i3c_slave_flags enumerators OR'd together to indicate the set of enabled interrupts.

static inline uint32_t I3C_SlaveGetPendingInterrupts(I3C_Type *base)

Returns the set of pending I3C slave interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of _i3c_slave_flags enumerators OR'd together to indicate the set of pending interrupts.

static inline void I3C_SlaveEnableDMA(I3C_Type *base, bool enableTx, bool enableRx, uint32_t width)

Enables or disables I3C slave DMA requests.

Parameters

- base – The I3C peripheral base address.
- enableTx – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- enableRx – Enable flag for receive DMA request. Pass true for enable, false for disable.
- width – DMA read/write unit in bytes.

static inline uint32_t I3C_SlaveGetTxFifoAddress(I3C_Type *base, uint32_t width)

Gets I3C slave transmit data register address for DMA transfer.

Parameters

- base – The I3C peripheral base address.

- width – DMA read/write unit in bytes.

Returns
The I3C Slave Transmit Data Register address.

static inline uint32_t I3C_SlaveGetRxFifoAddress(I3C_Type *base, uint32_t width)

Gets I3C slave receive data register address for DMA transfer.

Parameters

- base – The I3C peripheral base address.
- width – DMA read/write unit in bytes.

Returns
The I3C Slave Receive Data Register address.

static inline void I3C_SlaveSetWatermarks(I3C_Type *base, *i3c_tx_trigger_level_t* txLvl, *i3c_rx_trigger_level_t* rxLvl, bool flushTx, bool flushRx)

Sets the watermarks for I3C slave FIFOs.

Parameters

- base – The I3C peripheral base address.
- txLvl – Transmit FIFO watermark level. The kI3C_SlaveTxReadyFlag flag is set whenever the number of words in the transmit FIFO reaches *txLvl*.
- rxLvl – Receive FIFO watermark level. The kI3C_SlaveRxReadyFlag flag is set whenever the number of words in the receive FIFO reaches *rxLvl*.
- flushTx – true if TX FIFO is to be cleared, otherwise TX FIFO remains unchanged.
- flushRx – true if RX FIFO is to be cleared, otherwise RX FIFO remains unchanged.

static inline void I3C_SlaveGetFifoCounts(I3C_Type *base, size_t *rxCount, size_t *txCount)

Gets the current number of bytes in the I3C slave FIFOs.

Parameters

- base – The I3C peripheral base address.
- txCount – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.
- rxCount – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

status_t I3C_SlaveSend(I3C_Type *base, const void *txBuff, size_t txSize)

Performs a polling send transfer on the I3C bus.

Parameters

- base – The I3C peripheral base address.
- txBuff – The pointer to the data to be transferred.
- txSize – The length in bytes of the data to be transferred.

Returns
Error or success status returned by API.

status_t I3C_SlaveReceive(I3C_Type *base, void *rxBuff, size_t rxSize)

Performs a polling receive transfer on the I3C bus.

Parameters

- base – The I3C peripheral base address.
- rxBuff – The pointer to the data to be transferred.
- rxSize – The length in bytes of the data to be transferred.

Returns
Error or success status returned by API.

void I3C_SlaveTransferCreateHandle(I3C_Type *base, *i3c_slave_handle_t* *handle, *i3c_slave_transfer_callback_t* callback, void *userData)

Creates a new handle for the I3C slave non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the I3C_SlaveTransferAbort() API shall be called.

Note: The function also enables the NVIC IRQ for the input I3C. Need to notice that on some SoCs the I3C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- base – The I3C peripheral base address.
- handle – **[out]** Pointer to the I3C slave driver handle.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

status_t I3C_SlaveTransferNonBlocking(I3C_Type *base, *i3c_slave_handle_t* *handle, uint32_t eventMask)

Starts accepting slave transfers.

Call this API after calling I2C_SlaveInit() and I3C_SlaveTransferCreateHandle() to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to I3C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of i3c_slave_transfer_event_t enumerators for the events you wish to receive. The kI3C_SlaveTransmitEvent and kI3C_SlaveReceiveEvent events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the kI3C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

- base – The I3C peripheral base address.
- handle – Pointer to struct: _i3c_slave_handle structure which stores the transfer state.
- eventMask – Bit mask formed by OR'ing together i3c_slave_transfer_event_t enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and kI3C_SlaveAllEvents to enable all events.

Return values

- kStatus_Success – Slave transfers were successfully started.

- kStatus_I3C_Busy – Slave transfers have already been started on this handle.

status_t I3C_SlaveTransferGetCount(I3C_Type *base, *i3c_slave_handle_t* *handle, size_t *count)

Gets the slave transfer status during a non-blocking transfer.

Parameters

- base – The I3C peripheral base address.
- handle – Pointer to i2c_slave_handle_t structure.
- count – **[out]** Pointer to a value to hold the number of bytes transferred. May be NULL if the count is not required.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress –

void I3C_SlaveTransferAbort(I3C_Type *base, *i3c_slave_handle_t* *handle)

Aborts the slave non-blocking transfers.

Note: This API could be called at any time to stop slave for handling the bus events.

Parameters

- base – The I3C peripheral base address.
- handle – Pointer to struct: _i3c_slave_handle structure which stores the transfer state.

void I3C_SlaveTransferHandleIRQ(I3C_Type *base, void *intHandle)

Reusable routine to handle slave interrupts.

Note: This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

- base – The I3C peripheral base address.
- intHandle – Pointer to struct: _i3c_slave_handle structure which stores the transfer state.

enum _i3c_slave_flags

I3C slave peripheral flags.

The following status register flags can be cleared:

- kI3C_SlaveBusStartFlag
- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag

Only below flags can be enabled as interrupts.

- kI3C_SlaveBusStartFlag
- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag
- kI3C_SlaveRxReadyFlag

- kI3C_SlaveTxReadyFlag
- kI3C_SlaveDynamicAddrChangedFlag
- kI3C_SlaveReceivedCCCFlag
- kI3C_SlaveErrorFlag
- kI3C_SlaveHDRCommandMatchFlag
- kI3C_SlaveCCCHandledFlag
- kI3C_SlaveEventSentFlag

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI3C_SlaveNotStopFlag
Slave status not stop flag

enumerator kI3C_SlaveMessageFlag
Slave status message, indicating slave is listening to the bus traffic or responding

enumerator kI3C_SlaveRequiredReadFlag
Slave status required, either is master doing SDR read from slave, or is IBI pushing out.

enumerator kI3C_SlaveRequiredWriteFlag
Slave status request write, master is doing SDR write to slave, except slave in ENTDAA mode

enumerator kI3C_SlaveBusDAAFlag
I3C bus is in ENTDAA mode

enumerator kI3C_SlaveBusHDRModeFlag
I3C bus is in HDR mode

enumerator kI3C_SlaveBusStartFlag
Start/Re-start event is seen since the bus was last cleared

enumerator kI3C_SlaveMatchedFlag
Slave address(dynamic/static) matched since last cleared

enumerator kI3C_SlaveBusStopFlag
Stop event is seen since the bus was last cleared

enumerator kI3C_SlaveRxReadyFlag
Rx data ready in rx buffer flag

enumerator kI3C_SlaveTxReadyFlag
Tx buffer ready for Tx data flag

enumerator kI3C_SlaveDynamicAddrChangedFlag
Slave dynamic address has been assigned, re-assigned, or lost

enumerator kI3C_SlaveReceivedCCCFlag
Slave received Common command code

enumerator kI3C_SlaveErrorFlag
Error occurred flag

enumerator kI3C_SlaveHDRCommandMatchFlag
High data rate command match

enumerator kI3C_SlaveCCCHandledFlag
Slave received Common command code is handled by I3C module

enumerator kI3C_SlaveEventSentFlag
Slave IBI/P2P/MR/HJ event has been sent

enumerator kI3C_SlaveIbiDisableFlag
Slave in band interrupt is disabled.

enumerator kI3C_SlaveMasterRequestDisabledFlag
Slave master request is disabled.

enumerator kI3C_SlaveHotJoinDisabledFlag
Slave Hot-Join is disabled.

enumerator kI3C_SlaveClearFlags
All flags which are cleared by the driver upon starting a transfer.

enumerator kI3C_SlaveAllIrqFlags

enum _i3c_slave_error_flags
I3C slave error flags to indicate the causes.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI3C_SlaveErrorOverrunFlag
Slave internal from-bus buffer/FIFO overrun.

enumerator kI3C_SlaveErrorUnderrunFlag
Slave internal to-bus buffer/FIFO underrun

enumerator kI3C_SlaveErrorUnderrunNakFlag
Slave internal from-bus buffer/FIFO underrun and NACK error

enumerator kI3C_SlaveErrorTermFlag
Terminate error from master

enumerator kI3C_SlaveErrorInvalidStartFlag
Slave invalid start flag

enumerator kI3C_SlaveErrorSdrParityFlag
SDR parity error

enumerator kI3C_SlaveErrorHdrParityFlag
HDR parity error

enumerator kI3C_SlaveErrorHdrCRCFlag
HDR-DDR CRC error

enumerator kI3C_SlaveErrorS0S1Flag
S0 or S1 error

enumerator kI3C_SlaveErrorOverreadFlag
Over-read error

enumerator kI3C_SlaveErrorOverwriteFlag
Over-write error

enum _i3c_slave_event

I3C slave.event.

Values:

enumerator kI3C_SlaveEventNormal

Normal mode.

enumerator kI3C_SlaveEventIBI

In band interrupt event.

enumerator kI3C_SlaveEventMasterReq

Master request event.

enumerator kI3C_SlaveEventHotJoinReq

Hot-join event.

enum _i3c_slave_activity_state

I3C slave.activity state.

Values:

enumerator kI3C_SlaveNoLatency

Normal bus operation

enumerator kI3C_SlaveLatency1Ms

1ms of latency.

enumerator kI3C_SlaveLatency100Ms

100ms of latency.

enumerator kI3C_SlaveLatency10S

10s latency.

enum _i3c_slave_transfer_event

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to I3C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

Values:

enumerator kI3C_SlaveAddressMatchEvent

Received the slave address after a start or repeated start.

enumerator kI3C_SlaveTransmitEvent

Callback is requested to provide data to transmit (slave-transmitter role).

enumerator kI3C_SlaveReceiveEvent

Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator kI3C_SlaveRequiredTransmitEvent

Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator kI3C_SlaveStartEvent

A start/repeated start was detected.

enumerator kI3C_SlaveHDRCommandMatchEvent
Slave Match HDR Command.

enumerator kI3C_SlaveCompletionEvent
A stop was detected, completing the transfer.

enumerator kI3C_SlaveRequestSentEvent
Slave request event sent.

enumerator kI3C_SlaveReceivedCCCEvent
Slave received CCC event, need to handle by application.

enumerator kI3C_SlaveAllEvents
Bit mask of all available events.

typedef enum *_i3c_slave_event* i3c_slave_event_t
I3C slave.event.

typedef enum *_i3c_slave_activity_state* i3c_slave_activity_state_t
I3C slave.activity state.

typedef struct *_i3c_slave_config* i3c_slave_config_t
Structure with settings to initialize the I3C slave module.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

typedef enum *_i3c_slave_transfer_event* i3c_slave_transfer_event_t
Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to I3C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

typedef struct *_i3c_slave_transfer* i3c_slave_transfer_t
I3C slave transfer structure.

typedef struct *_i3c_slave_handle* i3c_slave_handle_t

typedef void (*i3c_slave_transfer_callback_t)(I3C_Type *base, *i3c_slave_transfer_t* *transfer, void *userData)

Slave event callback function pointer type.

This callback is used only for the slave non-blocking transfer API. To install a callback, use the I3C_SlaveSetCallback() function after you have created a handle.

Param base
Base address for the I3C instance on which the event occurred.

Param transfer
Pointer to transfer descriptor containing values passed to and/or from the callback.

Param userData
Arbitrary pointer-sized value passed from the application.

typedef void (*i3c_slave_isr_t)(I3C_Type *base, void *handle)

Typedef for slave interrupt handler.

struct i3c_slave_config

#include <fsl_i3c.h> Structure with settings to initialize the I3C slave module.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

bool enableSlave

Whether to enable slave.

uint8_t staticAddr

Static address.

uint16_t vendorID

Device vendor ID(manufacture ID).

uint32_t partNumber

Device part number info

uint8_t dcr

Device characteristics register information.

uint8_t bcr

Bus characteristics register information.

uint8_t hdrMode

Support hdr mode, could be OR logic in enumeration:i3c_hdr_mode_t.

bool nakAllRequest

Whether to reply NAK to all requests except broadcast CCC.

bool ignoreS0S1Error

Whether to ignore S0/S1 error in SDR mode.

bool offline

Whether to wait 60 us of bus quiet or HDR request to ensure slave track SDR mode safely.

bool matchSlaveStartStop

Whether to assert start/stop status only the time slave is addressed.

uint32_t maxWriteLength

Maximum write length.

uint32_t maxReadLength

Maximum read length.

struct i3c_slave_transfer

#include <fsl_i3c.h> I3C slave transfer structure.

Public Members

uint32_t event
: Reason the callback is being invoked.

uint8_t *txData
: Transfer buffer

size_t txDataSize
: Transfer size

uint8_t *rxData
: Transfer buffer

size_t rxDataSize
: Transfer size

status_t completionStatus
: Success or error code describing how the transfer completed. Only applies for kI3C_SlaveCompletionEvent.

size_t transferredCount
: Number of bytes actually transferred since start or last repeated start.

struct __i3c__slave__handle
: *#include <fsl_i3c.h>* I3C slave handle structure.

Note: The contents of this structure are private and subject to change.

Public Members

i3c_slave_transfer_t transfer
: I3C slave transfer copy.

bool isBusy
: Whether transfer is busy.

bool wasTransmit
: Whether the last transfer was a transmit.

uint32_t eventMask
: Mask of enabled events.

uint32_t transferredCount
: Count of bytes transferred.

i3c_slave_transfer_callback_t callback
: Callback function called at transfer event.

void *userData
: Callback parameter passed to callback.

size_t txFifoSize
: Tx Fifo size

2.28 IMU: Inter CPU Messaging Unit

status_t IMU_Init(imu_link_t link)

Initializes the IMU module.

This function sets IMU to initialized state, including:

- Flush the send FIFO.
- Unlock the send FIFO.
- Set the water mark to (IMU_MAX_MSG_FIFO_WATER_MARK)
- Flush the receive FIFO.

If IMU_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return kStatus_Timeout if flushing the receive FIFO takes too long.

Parameters

- link – IMU link.

Return values

- kStatus_Success – The IMU was initialized successfully.
- kStatus_InvalidArgument – Invalid link parameter.
- kStatus_Timeout – Timeout occurred while flushing the receive FIFO.

Returns
status_t

void IMU_Deinit(imu_link_t link)

De-initializes the IMU module.

Parameters

- link – IMU link.

static inline void IMU_WriteMsg(imu_link_t link, uint32_t msg)

Write one message to TX FIFO.

This function writes message to the TX FIFO, user need to make sure there is empty space in the TX FIFO, and TX FIFO not locked before calling this function.

Parameters

- link – IMU link.
- msg – The message to send.

static inline uint32_t IMU_ReadMsg(imu_link_t link)

Read one message from RX FIFO.

User need to make sure there is available message in the RX FIFO.

Parameters

- link – IMU link.

Returns
The message.

int32_t IMU_SendMsgsBlocking(imu_link_t link, const uint32_t *msgs, int32_t msgCount, bool lockSendFifo)

Blocking to send messages.

This function blocks until all messages have been filled to TX FIFO.

- If the TX FIFO is locked, this function returns IMU_ERR_TX_FIFO_LOCKED.
- If TX FIFO not locked, this function waits the available empty slot in TX FIFO, and fills the message to TX FIFO.
- To lock TX FIFO after filling all messages, set lockSendFifo to true.

If IMU_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return IMU_ERR_TIMEOUT if waiting for FIFO space takes too long.

Parameters

- link – IMU link.
- msgs – The messages to send.
- msgCount – Message count, one message is a 32-bit word.
- lockSendFifo – If set to true, the TX FIFO is locked after all messages filled to TX FIFO.

Returns

If TX FIFO is locked, this function returns IMU_ERR_TX_FIFO_LOCKED. If a timeout occurs while waiting for FIFO space, it returns IMU_ERR_TIMEOUT. Otherwise, this function returns the actual message count sent out, which equals msgCount because this function is blocking until all messages have been filled into TX FIFO or a timeout occurs.

int32_t IMU_TrySendMsgs(imu_link_t link, const uint32_t *msgs, int32_t msgCount, bool lockSendFifo)

Try to send messages.

This function is similar with IMU_SendMsgsBlocking, the difference is, this function tries to send as many as possible, if there is not enough empty slot in TX FIFO, this function fills messages to available empty slots and returns how many messages have been filled.

- If the TX FIFO is locked, this function returns IMU_ERR_TX_FIFO_LOCKED.
- If TX FIFO not locked, this function fills messages to TX FIFO empty slot, and returns how many messages have been filled.
- If lockSendFifo is set to true, TX FIFO is locked after all messages have been filled to TX FIFO. In other word, TX FIFO is locked if the function return value equals msgCount, when lockSendFifo set to true.

Parameters

- link – IMU link.
- msgs – The messages to send.
- msgCount – Message count, one message is a 32-bit word.
- lockSendFifo – If set to true, the TX FIFO is locked after all messages filled to TX FIFO.

Returns
If TX FIFO is locked, this function returns IMU_ERR_TX_FIFO_LOCKED, otherwise, this function returns the actual message count sent out.

int32_t IMU_TryReceiveMsgs(imu_link_t link, uint32_t *msgs, int32_t desiredMsgCount, bool *needAckLock)

Try to receive messages.

This function tries to read messages from RX FIFO. It reads the messages already exists in RX FIFO and returns the actual read count.

- If the RX FIFO has enough messages, this function reads the messages and returns.
- If the RX FIFO does not have enough messages, this function the messages in RX FIFO and returns the actual read count.
- During message reading, if RX FIFO is empty and locked, in this case peer CPU will not send message until current CPU send lock ack message. Then this function returns the message count actually received, and sets needAckLock to true to inform upper layer.

Parameters

- link – IMU link.
- msgs – The buffer to read messages.
- desiredMsgCount – Desired read count, one message is a 32-bit word.
- needAckLock – Upper layer should always check this value. When this is set to true by this function, upper layer should send lock ack message to peer CPU.

Returns
Count of messages actually received.

int32_t IMU_ReceiveMsgsBlocking(imu_link_t link, uint32_t *msgs, int32_t desiredMsgCount, bool *needAckLock)

Blocking to receive messages.

This function blocks until all desired messages have been received or the RX FIFO is locked.

- If the RX FIFO has enough messages, this function reads the messages and returns.
- If the RX FIFO does not have enough messages, this function waits for the new messages.
- During message reading, if RX FIFO is empty and locked, in this case peer CPU will not send message until current CPU send lock ack message. Then this function returns the message count actually received, and sets needAckLock to true to inform upper layer.

Parameters

- link – IMU link.
- msgs – The buffer to read messages.
- desiredMsgCount – Desired read count, one message is a 32-bit word.
- needAckLock – Upper layer should always check this value. When this is set to true by this function, upper layer should send lock ack message to peer CPU.

Returns
Count of messages actually received.

int32_t IMU_SendMsgPtrBlocking(imu_link_t link, uint32_t msgPtr, bool lockSendFifo)

Blocking to send messages pointer.

Compared with IMU_SendMsgsBlocking, this function fills message pointer to TX FIFO, but not the message content.

This function blocks until the message pointer is filled to TX FIFO.

- If the TX FIFO is locked, this function returns IMU_ERR_TX_FIFO_LOCKED.
- If TX FIFO not locked, this function waits the available empty slot in TX FIFO, and fills the message pointer to TX FIFO.
- To lock TX FIFO after filling the message pointer, set lockSendFifo to true.

If IMU_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return IMU_ERR_TIMEOUT if waiting for FIFO space takes too long.

Parameters

- link – IMU link.
- msgPtr – The buffer pointer to message to send.
- lockSendFifo – If set to true, the TX FIFO is locked after message pointer filled to TX FIFO.

Returns

If TX FIFO is locked, this function returns IMU_ERR_TX_FIFO_LOCKED. If a timeout occurs while waiting for FIFO space, it returns IMU_ERR_TIMEOUT. Otherwise, this function returns 0 to indicate success.

static inline void IMU_LockSendFifo(imu_link_t link, bool lock)

Lock or unlock the TX FIFO.

Parameters

- link – IMU link.
- lock – Use true to lock the FIFO, use false to unlock.

void IMU_FlushSendFifo(imu_link_t link)

Flush the send FIFO.

Flush all messages in send FIFO.

Parameters

- link – IMU link.

static inline void IMU_SetSendFifoWaterMark(imu_link_t link, uint8_t waterMark)

Set send FIFO warter mark.

The warter mark must be less than IMU_MAX_MSG_FIFO_WATER_MARK, i.e. 0 < waterMark <= IMU_MAX_MSG_FIFO_WATER_MARK.

Parameters

- link – IMU link.
- waterMark – Send FIFO warter mark.

static inline uint32_t IMU_GetReceivedMsgCount(imu_link_t link)

Get the message count in receive FIFO.

Parameters

- link – IMU link.

Returns
The message count in receive FIFO.

static inline uint32_t IMU_GetSendFifoEmptySpace(imu_link_t link)
Get the empty slot in send FIFO.

Parameters

- link – IMU link.

Returns
The empty slot count in send FIFO.

uint32_t IMU_GetStatusFlags(imu_link_t link)
Gets the IMU status flags.

Parameters

- link – IMU link.

Returns
Bit mask of the IMU status flags, see _imu_status_flags.

static inline void IMU_ClearPendingInterrupts(imu_link_t link, uint32_t mask)
Clear the IMU IRQ.

Parameters

- link – IMU link.
- mask – Bit mask of the interrupts to clear, see _imu_interrupts.

FSL_IMU_DRIVER_VERSION
IMU driver version.

enum _imu_status_flags
IMU status flags. .

Values:

enumerator kIMU_TxFifoEmpty

enumerator kIMU_TxFifoFull

enumerator kIMU_TxFifoAlmostFull

enumerator kIMU_TxFifoLocked

enumerator kIMU_RxFifoEmpty

enumerator kIMU_RxFifoFull

enumerator kIMU_RxFifoAlmostFull

enumerator kIMU_RxFifoLocked

enum _imu_interrupts
IMU interrupt. .

Values:

enumerator kIMU_RxMsgReadyInterrupt

enumerator kIMU_TxFifoSpaceAvailableInterrupt

IMU_MSG_FIFO_STATUS_MSG_FIFO_LOCKED_MASK

IMU_MSG_FIFO_STATUS_MSG_FIFO_ALMOST_FULL_MASK

IMU_MSG_FIFO_STATUS_MSG_FIFO_FULL_MASK

IMU_MSG_FIFO_STATUS_MSG_FIFO_EMPTY_MASK

IMU_MSG_FIFO_STATUS_MSG_COUNT_MASK

IMU_MSG_FIFO_STATUS_MSG_COUNT_SHIFT

IMU_MSG_FIFO_STATUS_WR_PTR_MASK

IMU_MSG_FIFO_STATUS_WR_PTR_SHIFT

IMU_MSG_FIFO_STATUS_RD_PTR_MASK

IMU_MSG_FIFO_STATUS_RD_PTR_SHIFT

IMU_MSG_FIFO_CNTL_MSG_RDY_INT_CLR_MASK

IMU_MSG_FIFO_CNTL_SP_AV_INT_CLR_MASK

IMU_MSG_FIFO_CNTL_FIFO_FLUSH_MASK

IMU_MSG_FIFO_CNTL_WAIT_FOR_ACK_MASK

IMU_MSG_FIFO_CNTL_FIFO_FULL_WATERMARK_MASK

IMU_MSG_FIFO_CNTL_FIFO_FULL_WATERMARK_SHIFT

IMU_MSG_FIFO_CNTL_FIFO_FULL_WATERMARK(x)

IMU_WR_MSG(link, msg)

IMU_RD_MSG(link)

IMU_RX_FIFO_LOCKED(link)

IMU_TX_FIFO_LOCKED(link)

IMU_TX_FIFO_ALMOST_FULL(link)

IMU_RX_FIFO_EMPTY(link)
Get Rx FIFO empty status.

IMU_LOCK_TX_FIFO(link)

IMU_UNLOCK_TX_FIFO(link)

IMU_RX_FIFO_MSG_COUNT(link)

IMU_TX_FIFO_MSG_COUNT(link)

IMU_RX_FIFO_MSG_COUNT_FROM_STATUS(rxFifoStatus)

IMU_RX_FIFO_LOCKED_FROM_STATUS(rxFifoStatus)

IMU_TX_FIFO_STATUS(link)

IMU_RX_FIFO_STATUS(link)

IMU_TX_FIFO_CNTL(link)

IMU_ERR_TX_FIFO_LOCKED
IMU driver returned error value.

IMU_ERR_TIMEOUT
IMU driver returned error value timeout.

IMU_MSG_FIFO_MAX_COUNT
Maximum message numbers in FIFO.

IMU_MAX_MSG_FIFO_WATER_MARK
Maximum message FIFO warter mark.

IMU_FIFO_SW_WRAPAROUND(ptr)

IMU_WR_PTR(link)

IMU_RD_PTR(link)

IMU_BUSY_POLL_COUNT
Maximum polling iterations for IMU waiting loops.

This parameter defines the maximum number of iterations for any polling loop in the IMU driver code before timing out and returning an error.

It applies to all waiting loops in IMU driver.

This is a count of loop iterations, not a time-based value.

If defined as 0, polling loops will continue indefinitely until their exit condition is met, which could potentially cause the system to hang if sensors don't respond or if communication interfaces fail.

struct IMU_Type
#include <fsl_imu.h> IMU register structure.

2.29 Common Driver

FSL_COMMON_DRIVER_VERSION
common driver version.

DEBUG_CONSOLE_DEVICE_TYPE_NONE
No debug console.

DEBUG_CONSOLE_DEVICE_TYPE_UART
Debug console based on UART.

DEBUG_CONSOLE_DEVICE_TYPE_LPUART
Debug console based on LPUART.

DEBUG_CONSOLE_DEVICE_TYPE_LPSCI
Debug console based on LPSCI.

DEBUG_CONSOLE_DEVICE_TYPE_USBCDC
Debug console based on USBCDC.

DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM
Debug console based on FLEXCOMM.

DEBUG_CONSOLE_DEVICE_TYPE_IUART
Debug console based on i.MX UART.

DEBUG_CONSOLE_DEVICE_TYPE_VUSART
Debug console based on LPC_VUSART.

DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART
Debug console based on LPC_USART.

DEBUG_CONSOLE_DEVICE_TYPE_SWO
Debug console based on SWO.

DEBUG_CONSOLE_DEVICE_TYPE_QSCI
Debug console based on QSCI.

MIN(a, b)
Computes the minimum of *a* and *b*.

MAX(a, b)
Computes the maximum of *a* and *b*.

UINT16_MAX
Max value of uint16_t type.

UINT32_MAX
Max value of uint32_t type.

SDK_ATOMIC_LOCAL_ADD(addr, val)
Add value *val* from the variable at address *address*.

SDK_ATOMIC_LOCAL_SUB(addr, val)
Subtract value *val* to the variable at address *address*.

SDK_ATOMIC_LOCAL_SET(addr, bits)
Set the bits specifiled by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR(addr, bits)
Clear the bits specifiled by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_TOGGLE(addr, bits)
Toggle the bits specifiled by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR_AND_SET(addr, clearBits, setBits)
For the variable at address *address*, clear the bits specifiled by *clearBits* and set the bits specifiled by *setBits*.

SDK_ATOMIC_LOCAL_COMPARE_AND_SET(addr, expected, newValue)
For the variable at address *address*, check whether the value equal to *expected*. If value same as *expected* then update *newValue* to address and return **true** , else return **false** .

SDK_ATOMIC_LOCAL_TEST_AND_SET(addr, newValue)
For the variable at address *address*, set as *newValue* value and return old value.

USEC_TO_COUNT(us, clockFreqInHz)
Macro to convert a microsecond period to raw count value

COUNT_TO_USEC(count, clockFreqInHz)
Macro to convert a raw count value to microsecond

MSEC_TO_COUNT(ms, clockFreqInHz)
Macro to convert a millisecond period to raw count value

COUNT_TO_MSEC(count, clockFreqInHz)
Macro to convert a raw count value to millisecond

SDK_ISR_EXIT_BARRIER

SDK_ALIGN(var, alignbytes)
Macro to define a variable with alignbytes alignment

SDK_SIZEALIGN(var, alignbytes)

Macro to define a variable with L1 d-cache line size alignment

Macro to define a variable with L2 cache line size alignment

Macro to change a value to a given size aligned value (rounded up)

SDK_SIZEALIGN_UP(var, alignbytes)

Macro to change a value to a given size aligned value (rounded up), the wrapper of SDK_SIZEALIGN

SDK_SIZEALIGN_DOWN(var, alignbytes)

Macro to change a value to a given size aligned value (rounded down)

SDK_IS_ALIGNED(var, alignbytes)

Macro to check if a value is aligned to a given size

AT_NONCACHEABLE_SECTION(var)

Define a variable *var*, and place it in non-cacheable section.

AT_NONCACHEABLE_SECTION_ALIGN(var, alignbytes)

Define a variable *var*, and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes*.

AT_NONCACHEABLE_SECTION_INIT(var)

Define a variable *var* with initial value, and place it in non-cacheable section.

AT_NONCACHEABLE_SECTION_ALIGN_INIT(var, alignbytes)

Define a variable *var* with initial value, and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes*.

AT_CACHE_LINE_SECTION(var)

Define a variable *var*, which is cache line size aligned and be placed in CacheLineData section.

AT_CACHE_LINE_SECTION_INIT(var)

Define a variable *var* with initial value, which is cache line size aligned and be placed in CacheLineData.init section.

AT_QUICKACCESS_SECTION_CODE(func)

Place function in a section which can be accessed quickly by core.

AT_QUICKACCESS_SECTION_DATA(var)

Place data in a section which can be accessed quickly by core.

AT_QUICKACCESS_SECTION_DATA_ALIGN(var, alignbytes)

Place data in a section which can be accessed quickly by core, and the variable address is set to align with *alignbytes*.

MCUX_RAMFUNC

Function attribute to place function in RAM. For example, to place function my_func in ram, use like:

```
MCUX_RAMFUNC my_func
```

RAMFUNCTION_SECTION_CODE(func)

Place function in ram.

enum _status_groups

Status group numbers.

Values:

enumerator kStatusGroup_Generic
Group number for generic status codes.

enumerator kStatusGroup_FLASH
Group number for FLASH status codes.

enumerator kStatusGroup_LPSPI
Group number for LPSPI status codes.

enumerator kStatusGroup_FLEXIO_SPI
Group number for FLEXIO SPI status codes.

enumerator kStatusGroup_DSPI
Group number for DSPI status codes.

enumerator kStatusGroup_FLEXIO_UART
Group number for FLEXIO UART status codes.

enumerator kStatusGroup_FLEXIO_I2C
Group number for FLEXIO I2C status codes.

enumerator kStatusGroup_LPI2C
Group number for LPI2C status codes.

enumerator kStatusGroup_UART
Group number for UART status codes.

enumerator kStatusGroup_I2C
Group number for UART status codes.

enumerator kStatusGroup_LPSCI
Group number for LPSCI status codes.

enumerator kStatusGroup_LPUART
Group number for LPUART status codes.

enumerator kStatusGroup_SPI
Group number for SPI status code.

enumerator kStatusGroup_XRDC
Group number for XRDC status code.

enumerator kStatusGroup_SEMA42
Group number for SEMA42 status code.

enumerator kStatusGroup_SDHC
Group number for SDHC status code

enumerator kStatusGroup_SDMMC
Group number for SDMMC status code

enumerator kStatusGroup_SAI
Group number for SAI status code

enumerator kStatusGroup_MCG
Group number for MCG status codes.

enumerator kStatusGroup_SCG
Group number for SCG status codes.

enumerator kStatusGroup_SDSPI
Group number for SDSPI status codes.

enumerator kStatusGroup_FLEXIO_I2S
Group number for FLEXIO I2S status codes

enumerator kStatusGroup_FLEXIO_MCULCD
Group number for FLEXIO LCD status codes

enumerator kStatusGroup_FLASHIAP
Group number for FLASHIAP status codes

enumerator kStatusGroup_FLEXCOMM_I2C
Group number for FLEXCOMM I2C status codes

enumerator kStatusGroup_I2S
Group number for I2S status codes

enumerator kStatusGroup_IUART
Group number for IUART status codes

enumerator kStatusGroup_CSI
Group number for CSI status codes

enumerator kStatusGroup_MIPI_DSI
Group number for MIPI DSI status codes

enumerator kStatusGroup_SDRAMC
Group number for SDRAMC status codes.

enumerator kStatusGroup_POWER
Group number for POWER status codes.

enumerator kStatusGroup_ENET
Group number for ENET status codes.

enumerator kStatusGroup_PHY
Group number for PHY status codes.

enumerator kStatusGroup_TRGMUX
Group number for TRGMUX status codes.

enumerator kStatusGroup_SMARTCARD
Group number for SMARTCARD status codes.

enumerator kStatusGroup_LMEM
Group number for LMEM status codes.

enumerator kStatusGroup_QSPI
Group number for QSPI status codes.

enumerator kStatusGroup_DMA
Group number for DMA status codes.

enumerator kStatusGroup_EDMA
Group number for EDMA status codes.

enumerator kStatusGroup_DMAMGR
Group number for DMAMGR status codes.

enumerator kStatusGroup_FLEXCAN
Group number for FlexCAN status codes.

enumerator kStatusGroup_LTC
Group number for LTC status codes.

enumerator kStatusGroup_FLEXIO_CAMERA
Group number for FLEXIO CAMERA status codes.

enumerator kStatusGroup_LPC_SPI
Group number for LPC_SPI status codes.

enumerator kStatusGroup_LPC_USART
Group number for LPC_USART status codes.

enumerator kStatusGroup_DMIC
Group number for DMIC status codes.

enumerator kStatusGroup_SDIF
Group number for SDIF status codes.

enumerator kStatusGroup_SPIFI
Group number for SPIFI status codes.

enumerator kStatusGroup_OTP
Group number for OTP status codes.

enumerator kStatusGroup_MCAN
Group number for MCAN status codes.

enumerator kStatusGroup_CAAM
Group number for CAAM status codes.

enumerator kStatusGroup_ECSPI
Group number for ECSPI status codes.

enumerator kStatusGroup_USDHC
Group number for USDHC status codes.

enumerator kStatusGroup_LPC_I2C
Group number for LPC_I2C status codes.

enumerator kStatusGroup_DCP
Group number for DCP status codes.

enumerator kStatusGroup_MSCAN
Group number for MSCAN status codes.

enumerator kStatusGroup_ESAI
Group number for ESAI status codes.

enumerator kStatusGroup_FLEXSPI
Group number for FLEXSPI status codes.

enumerator kStatusGroup_MMDC
Group number for MMDC status codes.

enumerator kStatusGroup_PDM
Group number for MIC status codes.

enumerator kStatusGroup_SDMA
Group number for SDMA status codes.

enumerator kStatusGroup_ICS
Group number for ICS status codes.

enumerator kStatusGroup_SPDIF
Group number for SPDIF status codes.

enumerator kStatusGroup_LPC_MINISPI
Group number for LPC_MINISPI status codes.

enumerator kStatusGroup_HASHCRYPT
Group number for Hashcrypt status codes

enumerator kStatusGroup_LPC_SPI_SSP
Group number for LPC_SPI_SSP status codes.

enumerator kStatusGroup_I3C
Group number for I3C status codes

enumerator kStatusGroup_LPC_I2C_1
Group number for LPC_I2C_1 status codes.

enumerator kStatusGroup_NOTIFIER
Group number for NOTIFIER status codes.

enumerator kStatusGroup_DebugConsole
Group number for debug console status codes.

enumerator kStatusGroup_SEMC
Group number for SEMC status codes.

enumerator kStatusGroup_ApplicationRangeStart
Starting number for application groups.

enumerator kStatusGroup_IAP
Group number for IAP status codes

enumerator kStatusGroup_SFA
Group number for SFA status codes

enumerator kStatusGroup_SPC
Group number for SPC status codes.

enumerator kStatusGroup_PUF
Group number for PUF status codes.

enumerator kStatusGroup_TOUCH_PANEL
Group number for touch panel status codes

enumerator kStatusGroup_VBAT
Group number for VBAT status codes

enumerator kStatusGroup_XSPI
Group number for XSPI status codes

enumerator kStatusGroup_PNGDEC
Group number for PNGDEC status codes

enumerator kStatusGroup_JPEGDEC
Group number for JPEGDEC status codes

enumerator kStatusGroup_AUDMIX
Group number for AUDMIX status codes

enumerator kStatusGroup_HAL_GPIO
Group number for HAL GPIO status codes.

enumerator kStatusGroup_HAL_UART
Group number for HAL UART status codes.

enumerator kStatusGroup_HAL_TIMER
Group number for HAL TIMER status codes.

enumerator kStatusGroup_HAL_SPI
Group number for HAL SPI status codes.

enumerator kStatusGroup_HAL_I2C
Group number for HAL I2C status codes.

enumerator kStatusGroup_HAL_FLASH
Group number for HAL FLASH status codes.

enumerator kStatusGroup_HAL_PWM
Group number for HAL PWM status codes.

enumerator kStatusGroup_HAL_RNG
Group number for HAL RNG status codes.

enumerator kStatusGroup_HAL_I2S
Group number for HAL I2S status codes.

enumerator kStatusGroup_HAL_ADC_SENSOR
Group number for HAL ADC SENSOR status codes.

enumerator kStatusGroup_TIMERMANAGER
Group number for TiMER MANAGER status codes.

enumerator kStatusGroup_SERIALMANAGER
Group number for SERIAL MANAGER status codes.

enumerator kStatusGroup_LED
Group number for LED status codes.

enumerator kStatusGroup_BUTTON
Group number for BUTTON status codes.

enumerator kStatusGroup_EXTERN_EEPROM
Group number for EXTERN EEPROM status codes.

enumerator kStatusGroup_SHELL
Group number for SHELL status codes.

enumerator kStatusGroup_MEM_MANAGER
Group number for MEM MANAGER status codes.

enumerator kStatusGroup_LIST
Group number for List status codes.

enumerator kStatusGroup_OSA
Group number for OSA status codes.

enumerator kStatusGroup_COMMON_TASK
Group number for Common task status codes.

enumerator kStatusGroup_MSG
Group number for messaging status codes.

enumerator kStatusGroup_SDK_OCOTP
Group number for OCOTP status codes.

enumerator kStatusGroup_SDK_FLEXSPINOR
Group number for FLEXSPINOR status codes.

enumerator kStatusGroup_CODEC
Group number for codec status codes.

enumerator kStatusGroup_ASRC
Group number for codec status ASRC.

enumerator kStatusGroup_OTFAD
Group number for codec status codes.

enumerator kStatusGroup_SDIOSLV
Group number for SDIOSLV status codes.

enumerator kStatusGroup_MECC
Group number for MECC status codes.

enumerator kStatusGroup_ENET_QOS
Group number for ENET_QOS status codes.

enumerator kStatusGroup_LOG
Group number for LOG status codes.

enumerator kStatusGroup_I3CBUS
Group number for I3CBUS status codes.

enumerator kStatusGroup_QSCI
Group number for QSCI status codes.

enumerator kStatusGroup_ELEMU
Group number for ELEMU status codes.

enumerator kStatusGroup_QUEUEDSPI
Group number for QSPI status codes.

enumerator kStatusGroup_POWER_MANAGER
Group number for POWER_MANAGER status codes.

enumerator kStatusGroup_IPED
Group number for IPED status codes.

enumerator kStatusGroup_ELS_PKC
Group number for ELS PKC status codes.

enumerator kStatusGroup_CSS_PKC
Group number for CSS PKC status codes.

enumerator kStatusGroup_HOSTIF
Group number for HOSTIF status codes.

enumerator kStatusGroup_CLIF
Group number for CLIF status codes.

enumerator kStatusGroup_BMA
Group number for BMA status codes.

enumerator kStatusGroup_NETC
Group number for NETC status codes.

enumerator kStatusGroup_ELE
Group number for ELE status codes.

enumerator kStatusGroup_GLIKEY
Group number for GLIKEY status codes.

enumerator kStatusGroup_AON_POWER
Group number for AON_POWER status codes.

enumerator kStatusGroup_AON_COMMON
Group number for AON_COMMON status codes.

enumerator kStatusGroup_ENDAT3
Group number for ENDAT3 status codes.

enumerator kStatusGroup_HIPERFACE
Group number for HIPERFACE status codes.

enumerator kStatusGroup_NPX
Group number for NPX status codes.

enumerator kStatusGroup_ELA_CSEC
Group number for ELA_CSEC status codes.

enumerator kStatusGroup_FLEXIO_T_FORMAT
Group number for T-format status codes.

enumerator kStatusGroup_FLEXIO_A_FORMAT
Group number for A-format status codes.

enumerator kStatusGroup_LPC_QSPI
Group number for LPC QSPI status codes.

Generic status return codes.

Values:

enumerator kStatus_Success
Generic status for Success.

enumerator kStatus_Fail
Generic status for Fail.

enumerator kStatus_ReadOnly
Generic status for read only failure.

enumerator kStatus_OutOfRange
Generic status for out of range access.

enumerator kStatus_InvalidArgument
Generic status for invalid argument check.

enumerator kStatus_Timeout
Generic status for timeout.

enumerator kStatus_NoTransferInProgress
Generic status for no transfer in progress.

enumerator kStatus_Busy
Generic status for module is busy.

enumerator kStatus_NoData
Generic status for no data is found for the operation.

typedef int32_t status_t
Type used for all status and error return values.

void *SDK_Malloc(size_t size, size_t alignbytes)

Allocate memory with given alignment and aligned size.

This is provided to support the dynamically allocated memory used in cache-able region.

Parameters

- size – The length required to malloc.
- alignbytes – The alignment size.

Return values

The – allocated memory.

void SDK_Free(void *ptr)

Free memory.

Parameters

- ptr – The memory to be release.

void SDK_DelayAtLeastUs(uint32_t delayTime_us, uint32_t coreClock_Hz)

Delay at least for some time. Please note that, this API uses while loop for delay, different run-time environments make the time not precise, if precise delay count was needed, please implement a new delay function with hardware timer.

Parameters

- delayTime_us – Delay time in unit of microsecond.
- coreClock_Hz – Core clock frequency with Hz.

static inline *status_t* EnableIRQ(IRQn_Type interrupt)

Enable specific interrupt.

Enable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only enables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ number.

Return values

- kStatus_Success – Interrupt enabled successfully
- kStatus_Fail – Failed to enable the interrupt

static inline *status_t* DisableIRQ(IRQn_Type interrupt)

Disable specific interrupt.

Disable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only disables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ number.

Return values

- kStatus_Success – Interrupt disabled successfully
- kStatus_Fail – Failed to disable the interrupt

static inline *status_t* EnableIRQWithPriority(IRQn_Type interrupt, uint8_t priNum)

Enable the IRQ, and also set the interrupt priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ to Enable.
- priNum – Priority number set to interrupt controller register.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline *status_t* IRQ_SetPriority(IRQn_Type interrupt, uint8_t priNum)

Set the IRQ priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ to set.
- priNum – Priority number set to interrupt controller register.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline *status_t* IRQ_ClearPendingIRQ(IRQn_Type interrupt)

Clear the pending IRQ flag.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The flag which IRQ to clear.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline uint32_t DisableGlobalIRQ(void)

Disable the global IRQ.

Disable the global interrupt and return the current primask register. User is required to provided the primask register for the EnableGlobalIRQ().

Returns
Current primask value.

static inline void EnableGlobalIRQ(uint32_t primask)

Enable the global IRQ.

Set the primask register with the provided primask value but not just enable the primask. The idea is for the convenience of integration of RTOS. some RTOS get its own management mechanism of primask. User is required to use the EnableGlobalIRQ() and DisableGlobalIRQ() in pair.

Parameters

- primask – value of primask register to be restored. The primask value is supposed to be provided by the DisableGlobalIRQ().

static inline bool _SDK_AtomicLocalCompareAndSet(uint32_t *addr, uint32_t expected, uint32_t newValue)

static inline uint32_t _SDK_AtomicTestAndSet(uint32_t *addr, uint32_t newValue)

FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ

Macro to use the default weak IRQ handler in drivers.

MAKE_STATUS(group, code)

Construct a status code value from a group and code number.

MAKE_VERSION(major, minor, bugfix)

Construct the version number for drivers.

The driver version is a 32-bit number, for both 32-bit platforms(such as Cortex M) and 16-bit platforms(such as DSC).

```
| Unused     || Major Version || Minor Version ||  Bug Fix     |
31        25  24          17  16          9  8           0
```

ARRAY_SIZE(x)

Computes the number of elements in an array.

UINT64_H(X)

Macro to get upper 32 bits of a 64-bit value

UINT64_L(X)

Macro to get lower 32 bits of a 64-bit value

SUPPRESS_FALL_THROUGH_WARNING()

For switch case code block, if case section ends without “break;” statement, there wil be fallthrough warning with compiler flag -Wextra or -Wimplicit-fallthrough=n when using armgcc. To suppress this warning, “SUPPRESS_FALL_THROUGH_WARNING();” need to be added at the end of each case section which misses “break;”statement.

MSDK_REG_SECURE_ADDR(x)

Convert the register address to the one used in secure mode.

MSDK_REG_NONSECURE_ADDR(x)

Convert the register address to the one used in non-secure mode.

MSDK_HAS_DWT_CYCCNT
The chip supports DWT CYCCNT or not.

MSDK_INVALID_IRQ_HANDLER
Invalid IRQ handler address.

2.30 LIN: Local Interconnect Network Driver

2.31 LIN Driver

uint32_t LIN_CalcMaxHeaderTimeoutCnt(uint32_t baudRate)

Calculates maximal header time lenght.

Theader_Maximum = 1.4 * THeader_Nominal, THeader_Nominal = 34 * TBit, (13 nominal bits of breack; 1 nominal bit of break delimiter; 10 bits for SYNC and 10 bits of PID) The function is not include time for conveying break and break delimiter TIME_OUT_UNIT is in micro second

Parameters

- baudRate – baudrate

uint32_t LIN_CalcMaxResTimeoutCnt(uint32_t baudRate, uint8_t size)

Calculates maximal header time length.

TResponse_Maximum = 1.4 * TResponse_Nominal, TResponse_Nominal = 10 * (NData+ 1) * TBit

Parameters

- baudRate – Baudrate
- size – Frame size

lin_status_t LIN_SetResponse(uint8_t instance, uint8_t *response_buff, uint8_t response_length, uint8_t max_frame_res_timeout)

Forwards a response to a lower level.

Parameters

- instance – LPUART instance
- response_buff – response message
- response_length – length of response
- max_frame_res_timeout – maximal timeout duration for message

Returns
An error code or lin_status_t

lin_status_t LIN_RxResponse(uint8_t instance, uint8_t *response_buff, uint8_t response_length, uint8_t max_frame_res_timeout)

Forwards a response to a higher level.

Parameters

- instance – LPUART instance
- response_buff – response message
- response_length – length of response
- max_frame_res_timeout – maximal timeout duration for message

Returns
An error code or lin_status_t

lin_status_t LIN_IgnoreResponse(uint8_t instance)

Put a node into idle state.

Parameters

- instance – LPUART instance

Returns
An error code or lin_status_t

void LIN_GetSlaveDefaultConfig(*lin_user_config_t* *linUserConfig)

Initializes linUserConfig variable for a slave node.

Parameters

- linUserConfig – Pointer to LIN user config structure

void LIN_GetMasterDefaultConfig(*lin_user_config_t* *linUserConfig)

Initializes linUserConfig variable for a master node.

Parameters

- linUserConfig – Pointer to LIN user config structure

void LIN_CalculateBaudrate(uint32_t instance, uint32_t baudRate_Bps, uint32_t srcClock_Hz, uint32_t *osr, uint16_t *sbr)

Calculates baudrate registers values for given baudrate.

Parameters

- instance – LPUART instance
- baudRate_Bps – LPUART baudrate
- srcClock_Hz – LPUART clock frequency
- osr – LPUART baudrate OSR value, return value
- sbr – LPUART baudrate SBR value, return value

void LIN_SetBaudrate(uint32_t instance, uint32_t osr, uint16_t sbr)

Set baudrate registers values.

Parameters

- instance – LPUART instance
- osr – LPUART baudrate OSR value
- sbr – LPUART baudrate SBR value

lin_status_t LIN_Init(uint32_t instance, *lin_user_config_t* *linUserConfig, *lin_state_t* *linCurrentState, uint32_t clockSource)

Initializes an instance LIN Hardware Interface for LIN Network.

The caller provides memory for the driver state structures during initialization. The user must select the LIN Hardware Interface clock source in the application to initialize the LIN Hardware Interface.

Parameters

- instance – LPUART instance
- linUserConfig – user configuration structure of type lin_user_config_t
- linCurrentState – pointer to the LIN Hardware Interface driver state structure

- clockSource – clock source frequency in Hz for the LIN Hardware Interface

Returns
An error code or lin_status_t

lin_status_t LIN_Deinit(uint32_t instance)

Shuts down the LIN Hardware Interface by disabling interrupts and transmitter/receiver.

Parameters

- instance – LPUART instance

Returns
An error code or lin_status_t

lin_callback_t LIN_InstallCallback(uint32_t instance, *lin_callback_t* function)

Installs callback function that is used for LIN_DRV_IRQHandler.

Note: After a callback is installed, it bypasses part of the LIN Hardware Interface IRQHandler logic. Therefore, the callback needs to handle the indexes of txBuff and txSize.

Parameters

- instance – LPUART instance.
- function – the LIN receive callback function.

Returns
Former LIN callback function pointer.

lin_status_t LIN_SendFrameDataBlocking(uint32_t instance, const uint8_t *txBuff, uint8_t txSize, uint32_t timeoutMSec)

Sends Frame data out through the LIN Hardware Interface using blocking method. This function will calculate the checksum byte and send it with the frame data. Blocking means that the function does not return until the transmission is complete.

Parameters

- instance – LPUART instance
- txBuff – source buffer containing 8-bit data chars to send
- txSize – the number of bytes to send
- timeoutMSec – timeout value in milli seconds

Returns
An error code or lin_status_t LIN_BUS_BUSY if the bus is currently busy, transmission cannot be started.

lin_status_t LIN_SendFrameData(uint32_t instance, const uint8_t *txBuff, uint8_t txSize)

Sends frame data out through the LIN Hardware Interface using non-blocking method. This enables an a-sync method for transmitting data. Non-blocking means that the function returns immediately. The application has to get the transmit status to know when the transmit is complete. This function will calculate the checksum byte and send it with the frame data.

Note: If users use LIN_TimeoutService in a timer interrupt handler, then before using this function, users have to set timeout counter to an appropriate value by using LIN_SetTimeoutCounter(instance, timeoutValue). The timeout value should be big enough to complete the transmission. Timeout in real time is (timeoutValue) * (time period that

LIN_TimeoutService is called). For example, if LIN_TimeoutService is called in an timer interrupt with period of 500 micro seconds, then timeout in real time is timeoutValue * 500 micro seconds.

Parameters

- instance – LPUART instance
- txBuff – source buffer containing 8-bit data chars to send
- txSize – the number of bytes to send

Returns

An error code or lin_status_t LIN_BUS_BUSY if the bus is currently busy, transmission cannot be started. LIN_SUCCESS if the transmission was completed.

lin_status_t LIN_GetTransmitStatus(uint32_t instance, uint8_t *bytesRemaining)

Get status of an on-going non-blocking transmission While sending frame data using non-blocking method, users can use this function to get status of that transmission. The bytesRemaining shows number of bytes that still needed to transmit.

Parameters

- instance – LPUART instance
- bytesRemaining – Number of bytes still needed to transmit

Returns

lin_status_t LIN_TX_BUSY if the transmission is still in progress. LIN_TIMEOUT if timeout occurred and transmission was not completed. LIN_SUCCESS if the transmission was successful.

lin_status_t LIN_ReceiveFrameDataBlocking(uint32_t instance, uint8_t *rxBuff, uint8_t rxSize, uint32_t timeoutMSec)

Receives frame data through the LIN Hardware Interface using blocking method. This function will check the checksum byte. If the checksum is correct, it will receive the frame data. Blocking means that the function does not return until the reception is complete.

Parameters

- instance – LPUART instance
- rxBuff – buffer containing 8-bit received data
- rxSize – the number of bytes to receive
- timeoutMSec – timeout value in milli seconds

Returns

An error code or lin_status_t

lin_status_t LIN_ReceiveFrameData(uint32_t instance, uint8_t *rxBuff, uint8_t rxSize)

Receives frame data through the LIN Hardware Interface using non- blocking method. This function will check the checksum byte. If the checksum is correct, it will receive it with the frame data. Non-blocking means that the function returns immediately. The application has to get the receive status to know when the reception is complete.

Note: If users use LIN_TimeoutService in a timer interrupt handler, then before using this function, users have to set timeout counter to an appropriate value by using LIN_SetTimeoutCounter(instance, timeoutValue). The timeout value should be big enough to complete the reception. Timeout in real time is (timeoutValue) * (time period that LIN_TimeoutService is called). For example, if LIN_TimeoutService is called in an timer

interrupt with period of 500 micro seconds, then timeout in real time is timeoutValue * 500 micro seconds.

Parameters

- instance – LPUART instance
- rxBuff – buffer containing 8-bit received data
- rxSize – the number of bytes to receive

Returns
An error code or lin_status_t

lin_status_t LIN_AbortTransferData(uint32_t instance)

Aborts an on-going non-blocking transmission/reception. While performing a non-blocking transferring data, users can call this function to terminate immediately the transferring.

Parameters

- instance – LPUART instance

Returns
An error code or lin_status_t

lin_status_t LIN_GetReceiveStatus(uint32_t instance, uint8_t *bytesRemaining)

Get status of an on-going non-blocking reception While receiving frame data using non-blocking method, users can use this function to get status of that receiving. This function return the current event ID, LIN_RX_BUSY while receiving and return LIN_SUCCESS, or timeout (LIN_TIMEOUT) when the reception is complete. The bytesRemaining shows number of bytes that still needed to receive.

Parameters

- instance – LPUART instance
- bytesRemaining – Number of bytes still needed to receive

Returns
lin_status_t LIN_RX_BUSY, LIN_TIMEOUT or LIN_SUCCESS

lin_status_t LIN_GoToSleepMode(uint32_t instance)

Puts current LIN node to sleep mode This function changes current node state to LIN_NODE_STATE_SLEEP_MODE.

Parameters

- instance – LPUART instance

Returns
An error code or lin_status_t

lin_status_t LIN_GotoIdleState(uint32_t instance)

Puts current LIN node to Idle state This function changes current node state to LIN_NODE_STATE_IDLE.

Parameters

- instance – LPUART instance

Returns
An error code or lin_status_t

lin_status_t LIN_SendWakeupSignal(uint32_t instance)

Sends a wakeup signal through the LIN Hardware Interface.

Parameters

- instance – LPUART instance

Returns
An error code or lin_status_t

lin_node_state_t LIN_GetCurrentNodeState(uint32_t instance)

Get the current LIN node state.

Parameters

- instance – LPUART instance

Returns
current LIN node state

void LIN_TimeoutService(uint32_t instance)

Callback function for Timer Interrupt Handler Users may use (optional, not required) LIN_TimeoutService to check if timeout has occurred during non-blocking frame data transmission and reception. User may initialize a timer (for example FTM) in Output Compare Mode with period of 500 micro seconds (recommended). In timer IRQ handler, call this function.

Parameters

- instance – LPUART instance

void LIN_SetTimeoutCounter(uint32_t instance, uint32_t timeoutValue)

Set Value for Timeout Counter that is used in LIN_TimeoutService.

Parameters

- instance – LPUART instance
- timeoutValue – Timeout Value to be set

lin_status_t LIN_MasterSendHeader(uint32_t instance, uint8_t id)

Sends frame header out through the LIN Hardware Interface using a non-blocking method. This function sends LIN Break field, sync field then the ID with correct parity.

Parameters

- instance – LPUART instance
- id – Frame Identifier

Returns
An error code or lin_status_t

lin_status_t LIN_EnableIRQ(uint32_t instance)

Enables LIN hardware interrupts.

Parameters

- instance – LPUART instance

Returns
An error code or lin_status_t

lin_status_t LIN_DisableIRQ(uint32_t instance)

Disables LIN hardware interrupts.

Parameters

- instance – LPUART instance

Returns
An error code or lin_status_t

void LIN_IRQHandler(uint8_t instance)

Interrupt handler for LIN Hardware Interface.

Parameters

- instance – LPUART instance

lin_status_t LIN_AutoBaudCapture(uint32_t instance)

This function capture bits time to detect break char, calculate baudrate from sync bits and enable transceiver if autobaud successful. This function should only be used in Slave. The timer should be in mode input capture of both rising and falling edges. The timer input capture pin should be externally connected to RXD pin.

Parameters

- instance – LPUART instance

Returns

lin_status_t

uint8_t LIN_ProcessParity(uint8_t PID, uint8_t typeAction)

Makes or checks parity bits. If action is checking parity, the function returns ID value if parity bits are correct or 0xFF if parity bits are incorrect. If action is making parity bits, then from input value of ID, the function returns PID. This is not a public API as it is called by other API functions.

Parameters

- PID – PID byte in case of checking parity bits or ID byte in case of making parity bits.
- typeAction – 1 for Checking parity bits, 0 for making parity bits

Returns

0xFF if parity bits are incorrect, ID in case of checking parity bits and they are correct. Function returns PID in case of making parity bits.

uint8_t LIN_MakeChecksumByte(const uint8_t *buffer, uint8_t sizeBuffer, uint8_t PID)

Makes the checksum byte for a frame.

Parameters

- buffer – Pointer to Tx buffer
- sizeBuffer – Number of bytes that are contained in the buffer.
- PID – Protected Identifier byte.

Returns

the checksum byte.

FSL_LIN_DRIVER_VERSION

LIN driver version.

enum lin_frame_type

List of LIN frame types.

Values:

enumerator LIN_FRM_UNCD

unconditional frame

enumerator LIN_FRM_EVNT

event triggered frame

enumerator LIN_FRM_SPRDC

sporadic frame

enumerator LIN_FRM_DIAG
diagnostic frame

enum lin_frame_response
List of LIN frame response types.

Values:

enumerator LIN_RES_NOTHING
node is neither publisher nor subscriber

enumerator LIN_RES_PUB
node is publisher

enumerator LIN_RES_SUB
node is subscriber

enum lin_diagnostic_mode
Diagnostic mode.

Values:

enumerator DIAG_NONE
None

enumerator DIAG_INTERLEAVE_MODE
Interleave mode

enumerator DIAG_ONLY_MODE
Diagnostic only mode

enum lin_event_id_t
Defines types for an enumerating event related to an Identifier.

Values:

enumerator LIN_NO_EVENT
No event yet

enumerator LIN_WAKEUP_SIGNAL
Received a wakeup signal

enumerator LIN_BAUDRATE_ADJUSTED
Indicate that baudrate was adjusted to Master's baudrate

enumerator LIN_RECV_BREAK_FIELD_OK
Indicate that correct Break Field was received

enumerator LIN_SYNC_OK
Sync byte is correct

enumerator LIN_SYNC_ERROR
Sync byte is incorrect

enumerator LIN_PID_OK
PID correct

enumerator LIN_PID_ERROR
PID incorrect

enumerator LIN_FRAME_ERROR
Framing Error

enumerator LIN_READBACK_ERROR
Readback data is incorrect

enumerator LIN_CHECKSUM_ERROR
Checksum byte is incorrect

enumerator LIN_TX_COMPLETED
Sending data completed

enumerator LIN_RX_COMPLETED
Receiving data completed

enumerator LIN_NO_DATA_TIMEOUT
No data timeout

enumerator LIN_BUS_ACTIVITY_TIMEOUT
Bus activity timeout

enumerator LIN_TIMEOUT_ERROR
Indicate that timeout has occurred

enumerator LIN_LAST_RESPONSE_SHORT_ERROR
Indicate that the last frame was too short

enum lin_status_t
Defines Error codes of the LIN driver.

Values:

enumerator LIN_IFC_NOT_SUPPORT
This interface is not supported

enumerator LIN_INITIALIZED
LIN Hardware has been initialized

enumerator LIN_SUCCESS
Successfully done

enumerator LIN_ERROR
Error

enumerator LIN_TX_BUSY
Transmitter is busy

enumerator LIN_RX_BUSY
Receiver is busy

enumerator LIN_BUS_BUSY
Bus is busy

enumerator LIN_NO_TRANSFER_IN_PROGRESS
No data transfer is in progress

enumerator LIN_TIMEOUT
Timeout

enumerator LIN_LPUART_STAT_CLOCK_GATED_OFF
LPUART is gated from clock manager

enum lin_node_state_t
Define type for an enumerating LIN Node state.

Values:

enumerator LIN_NODE_STATE_UNINIT
Uninitialized state

enumerator LIN_NODE_STATE_SLEEP_MODE
Sleep mode state

enumerator LIN_NODE_STATE_IDLE
Idle state

enumerator LIN_NODE_STATE_SEND_BREAK_FIELD
Send break field state

enumerator LIN_NODE_STATE_RECV_SYNC
Receive the synchronization byte state

enumerator LIN_NODE_STATE_SEND_PID
Send PID state

enumerator LIN_NODE_STATE_RECV_PID
Receive PID state

enumerator LIN_NODE_STATE_RECV_DATA
Receive data state

enumerator LIN_NODE_STATE_RECV_DATA_COMPLETED
Receive data completed state

enumerator LIN_NODE_STATE_SEND_DATA
Send data state

enumerator LIN_NODE_STATE_SEND_DATA_COMPLETED
Send data completed state

enum lin_protocol_handle
List of protocols.

Values:

enumerator LIN_PROTOCOL_21
LIN protocol version 2.1

enumerator LIN_PROTOCOL_J2602
J2602 protocol

enum lin_supported_baudrates_t
List of supported baudrates for autobaud feature.

Values:

enumerator kLIN_BAUD_2400

enumerator kLIN_BAUD_4800

enumerator kLIN_BAUD_9600

enumerator kLIN_BAUD_14400

enumerator kLIN_BAUD_19200

typedef void (*lin_timer_get_time_interval_t)(uint32_t *nanoSeconds)
Callback function to get time interval in nano seconds.

typedef void (*lin_callback_t)(uint32_t instance, void *linState)
LIN Driver callback function type.

LPUART_Type *const g_linLpuartBase[1]
Table of base addresses for LPUART instances.

const IRQn_Type g_linLpuartRxTxIrqId[1]
Table to save LPUART IRQ enumeration numbers defined in the CMSIS header file.

lin_baudrate_values_t g_linConfigBaudrates[5U]
Table to save LIN user config buadrate values.

lin_state_t *g_linStatePtr[1]
Pointers to LPUART bases for each instance.

Table to save LPUART state structure pointers

lin_user_config_t *g_linUserconfigPtr[1]
Table to save LIN user config structure pointers.

LIN_SLAVE

LIN_MASTER

MAKE_PARITY

CHECK_PARITY

LIN_TIME_OUT_UNIT_US

LIN_MAKE_UNCONDITIONAL_FRAME

LIN_UPDATE_UNCONDITIONAL_FRAME

LIN_NUM_OF_SUPP_BAUDRATES

struct lin_user_config_t
#include <fsl_lin.h> LIN hardware configuration structure.

Public Members

uint8_t hardware_instance
interface instance number

uint32_t baudRate
baudrate of LIN Hardware Interface to configure

bool nodeFunction
Node function as Master or Slave

bool autobaudEnable
Enable Autobaud feature

lin_timer_get_time_interval_t timerGetTimeIntervalCallback
Callback function to get time interval in nano seconds

struct lin_state_t
#include <fsl_lin.h> Runtime state of the LIN driver.

Note that the caller provides memory for the driver state structures during initialization because the driver does not statically allocate memory.

Public Members

const uint8_t *txBuff
The buffer of data being sent.

uint8_t *rxBuff
The buffer of received data.

uint8_t cntByte
To count number of bytes already transmitted or received.

volatile uint8_t txSize
The remaining number of bytes to be received.

volatile uint8_t rxSize
The remaining number of bytes to be received.

uint8_t checkSum
Checksum byte.

volatile bool isTxBusy
True if the LIN interface is transmitting frame data.

volatile bool isRxBusy
True if the LIN interface is receiving frame data.

volatile bool isBusBusy
True if there are data, frame headers being transferred on bus

volatile bool isTxBlocking
True if transmit is blocking transaction.

volatile bool isRxBlocking
True if receive is blocking transaction.

lin_callback_t Callback
Callback function to invoke after receiving a byte or transmitting a byte.

uint8_t currentId
Current ID

uint8_t currentPid
Current PID

volatile *lin_event_id_t* currentEventId
Current ID Event

volatile *lin_node_state_t* currentNodeState
Current Node state

volatile uint32_t timeoutCounter
Value of the timeout counter

volatile bool timeoutCounterFlag
Timeout counter flag

volatile bool baudrateEvalEnable
Baudrate Evaluation Process Enable

volatile uint8_t fallingEdgeInterruptCount
Falling Edge count of a sync byte

uint32_t linSourceClockFreq
Frequency of the source clock for LIN

volatile bool txCompleted
Used to wait for LIN interface ISR to complete transmission.

volatile bool rxCompleted
Used to wait for LIN interface ISR to complete reception

struct lin_baudrate_values_t
#include <fsl_lin.h> Structure of baudrate properties.

struct lin_frame_struct
#include <fsl_lin.h> Informations of frame.

Public Members

lin_frame_type frm_type
Frame information (unconditional or event triggered..)

uint8_t frm_len
Length of the frame

lin_frame_response frm_response
Action response when received PID

uint8_t frm_offset
Frame byte offset in frame buffer

uint8_t flag_offset
Flag byte offset in flag buffer

uint8_t flag_size
Flag size in flag buffer

uint32_t delay
Frame delay

const uint8_t *frame_data_ptr
List of Signal to which the frame is associated and its offset

struct lin_protocol_user_config_t
#include <fsl_lin.h> Protocol configuration structure.

Public Members

lin_protocol_handle protocol_version
Protocol version

lin_protocol_handle language_version
Language version

uint8_t number_of_configurable_frames
Number of frame except diagnostic frames

uint8_t frame_start
Start index of frame list

const *lin_frame_struct* *frame_tbl_ptr
Frame list except diagnostic frames

const uint16_t *list_identifiers_ROM_ptr
Configuration in ROM

uint8_t *list_identifiers_RAM_ptr
Configuration in RAM

uint16_t max_idle_timeout_cnt
Max Idle timeout counter

uint16_t max_message_length
Max message length

struct lin_product_id
#include <fsl_lin.h> Product id structure.

Public Members

uint16_t supplier_id
Supplier ID

uint16_t function_id
Function ID

uint8_t variant
Variant value

struct lin_serial_number
#include <fsl_lin.h> Serial number.

Public Members

uint8_t serial_0
Serial 0

uint8_t serial_1
Serial 1

uint8_t serial_2
Serial 2

uint8_t serial_3
Serial 3

struct lin_protocol_state_t
#include <fsl_lin.h> Protocol state structure.

Public Members

uint16_t baud_rate
Adjusted baud rate

uint8_t *response_buffer_ptr
Response buffer

uint8_t response_length
Response length

uint8_t successful_transfer
Sets when frame is transferred successfully

uint8_t error_in_response
Sets when frame received/transmitter by the node contains an error in the response field

uint8_t timeout_in_response
Timeout response

bool go_to_sleep_flg
Go to sleep flag

uint8_t current_id
Current PID

uint8_t num_of_processed_frame
Number of processed frames

uint8_t num_of_successfull_frame
Number of processed frames

uint8_t next_transmit_tick
Used to count the next transmit tick

bool save_config_flg
Set when save configuration request has been received

lin_diagnostic_mode diagnostic_mode
Diagnostic mode

uint16_t frame_timeout_cnt
Frame timeout counter

uint16_t idle_timeout_cnt
Idle timeout counter, node will go to sleep when count down to 0

bool transmit_error_resp_sig_flg
Flag indicates that the error response signal is going to be sent

bool event_trigger_collision_flg
Flag indicates collision on bus

struct lin_node_attribute
#include <fsl_lin.h> Attributes of LIN node.

Public Members

uint8_t *configured_NAD_ptr
NAD value used in configuration command

uint8_t initial_NAD
Initial NAD

lin_product_id product_id
Product ID

lin_serial_number serial_number
Serial number

uint8_t *resp_err_frm_id_ptr
Pointer to the list of index of frames that carries response error signal

uint8_t num_frame_have_esignal
The count of frames that carry response error signal

uint8_t response_error_byte_offset
Byte offset of response error signal

uint8_t response_error_bit_offset
Bit offset of response error signal

uint8_t num_of_fault_state_signal
Number of Fault state signal

uint16_t P2_min
P2 min

uint16_t ST_min
ST min

uint16_t N_As_timeout
N_As_timeout

uint16_t N_Cr_timeout
N_Cr_timeout

struct lin_master_data_t
#include <fsl_lin.h> LIN master data structure.

Public Members

uint8_t active_schedule_id
Active schedule table id

uint8_t previous_schedule_id
Previous schedule table id

uint8_t *schedule_start_entry_ptr
Start entry of each schedule table

uint8_t data_buffer[8]
Master data buffer

2.32 LIN LPUART Driver

FSL_LIN_LPUART_DRIVER_VERSION
LIN LPUART driver version.

enum _lin_lpuart_stop_bit_count
Values:

enumerator kLPUART_OneStopBit
One stop bit

enumerator kLPUART_TwoStopBit
Two stop bits

enum _lin_lpuart_flags
Values:

enumerator kLPUART_TxDataRegEmptyFlag
Transmit data register empty flag, sets when transmit buffer is empty

enumerator kLPUART_TransmissionCompleteFlag
Transmission complete flag, sets when transmission activity complete

enumerator kLPUART_RxDataRegFullFlag
Receive data register full flag, sets when the receive data buffer is full

enumerator kLPUART_IdleLineFlag
Idle line detect flag, sets when idle line detected

enumerator kLPUART_RxOverrunFlag
Receive Overrun, sets when new data is received before data is read from receive register

enumerator kLPUART_NoiseErrorFlag
Receive takes 3 samples of each received bit. If any of these samples differ, noise flag sets

enumerator kLPUART_FramingErrorFlag
Frame error flag, sets if logic 0 was detected where stop bit expected

enumerator kLPUART_ParityErrorFlag
If parity enabled, sets upon parity error detection

enumerator kLPUART_LinBreakFlag
LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled

enumerator kLPUART_RxActiveEdgeFlag
Receive pin active edge interrupt flag, sets when active edge detected

enumerator kLPUART_RxActiveFlag
Receiver Active Flag (RAF), sets at beginning of valid start bit

enumerator kLPUART_DataMatch1Flag
The next character to be read from LPUART_DATA matches MA1

enumerator kLPUART_DataMatch2Flag
The next character to be read from LPUART_DATA matches MA2

enumerator kLPUART_NoiseErrorInRxDataRegFlag
NOISY bit, sets if noise detected in current data word

enumerator kLPUART_ParityErrorInRxDataRegFlag
PARITY bit, sets if noise detected in current data word

enumerator kLPUART_TxFifoEmptyFlag
TXEMPT bit, sets if transmit buffer is empty

enumerator kLPUART_RxFifoEmptyFlag
RXEMPT bit, sets if receive buffer is empty

enumerator kLPUART_TxFifoOverflowFlag
TXOF bit, sets if transmit buffer overflow occurred

enumerator kLPUART_RxFifoUnderflowFlag
RXUF bit, sets if receive buffer underflow occurred

enum _lin_lpuart_interrupt_enable
Values:

enumerator kLPUART_LinBreakInterruptEnable
LIN break detect.

enumerator kLPUART_RxActiveEdgeInterruptEnable
Receive Active Edge.

enumerator kLPUART_TxDataRegEmptyInterruptEnable
Transmit data register empty.

enumerator kLPUART_TransmissionCompleteInterruptEnable
Transmission complete.

enumerator kLPUART_RxDataRegFullInterruptEnable
Receiver data register full.

enumerator kLPUART_IdleLineInterruptEnable
Idle line.

enumerator kLPUART_RxOverrunInterruptEnable
Receiver Overrun.

enumerator kLPUART_NoiseErrorInterruptEnable
Noise error flag.

enumerator kLPUART_FramingErrorInterruptEnable
Framing error flag.

enumerator kLPUART_ParityErrorInterruptEnable
Parity error flag.

enumerator kLPUART_TxFifoOverflowInterruptEnable
Transmit FIFO Overflow.

enumerator kLPUART_RxFifoUnderflowInterruptEnable
Receive FIFO Underflow.

enum _lin_lpuart_status
Values:

enumerator kStatus_LPUART_TxBusy
TX busy

enumerator kStatus_LPUART_RxBusy
RX busy

enumerator kStatus_LPUART_TxIdle
LPUART transmitter is idle.

enumerator kStatus_LPUART_RxIdle
LPUART receiver is idle.

enumerator kStatus_LPUART_TxWatermarkTooLarge
TX FIFO watermark too large

enumerator kStatus_LPUART_RxWatermarkTooLarge
RX FIFO watermark too large

enumerator kStatus_LPUART_FlagCannotClearManually
Some flag can't manually clear

enumerator kStatus_LPUART_Error
Error happens on LPUART.

enumerator kStatus_LPUART_RxRingBufferOverrun
LPUART RX software ring buffer overrun.

enumerator kStatus_LPUART_RxHardwareOverrun
LPUART RX receiver overrun.

enumerator kStatus_LPUART_NoiseError
LPUART noise error.

enumerator kStatus_LPUART_FramingError
LPUART framing error.

enumerator kStatus_LPUART_ParityError
LPUART parity error.

enum lin_lpuart_bit_count_per_char_t
Values:

enumerator LPUART_8_BITS_PER_CHAR
8-bit data characters

enumerator LPUART_9_BITS_PER_CHAR
9-bit data characters

enumerator LPUART_10_BITS_PER_CHAR
10-bit data characters

typedef enum *_lin_lpuart_stop_bit_count* lin_lpuart_stop_bit_count_t

static inline bool LIN_LPUART_GetRxDataPolarity(const LPUART_Type *base)

static inline void LIN_LPUART_SetRxDataPolarity(LPUART_Type *base, bool polarity)

static inline void LIN_LPUART_WriteByte(LPUART_Type *base, uint8_t data)

static inline void LIN_LPUART_ReadByte(const LPUART_Type *base, uint8_t *readData)

status_t LIN_LPUART_CalculateBaudRate(LPUART_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz, uint32_t *osr, uint16_t *sbr)

Calculates the best osr and sbr value for configured baudrate.

Parameters

- base – LPUART peripheral base address
- baudRate_Bps – user configuration structure of type lin_user_config_t
- srcClock_Hz – pointer to the LIN_LPUART driver state structure
- osr – pointer to osr value
- sbr – pointer to sbr value

Returns
An error code or lin_status_t

void LIN_LPUART_SetBaudRate(LPUART_Type *base, uint32_t *osr, uint16_t *sbr)

Configure baudrate according to osr and sbr value.

Parameters

- base – LPUART peripheral base address
- osr – pointer to osr value
- sbr – pointer to sbr value

lin_status_t LIN_LPUART_Init(LPUART_Type *base, *lin_user_config_t* *linUserConfig, *lin_state_t* *linCurrentState, uint32_t linSourceClockFreq)

Initializes an LIN_LPUART instance for LIN Network.

The caller provides memory for the driver state structures during initialization. The user must select the LIN_LPUART clock source in the application to initialize the LIN_LPUART. This function initializes a LPUART instance for operation. This function will initialize the run-time state structure to keep track of the on-going transfers, initialize the module to user defined settings and default settings, set break field length to be 13 bit times minimum, enable the break detect interrupt, Rx complete interrupt, frame error detect interrupt, and enable the LPUART module transmitter and receiver

Parameters

- base – LPUART peripheral base address
- linUserConfig – user configuration structure of type lin_user_config_t
- linCurrentState – pointer to the LIN_LPUART driver state structure
- linSourceClockFreq – LIN source clock frequency in Hz

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_Deinit(LPUART_Type *base)

Shuts down the LIN_LPUART by disabling interrupts and transmitter/receiver.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_SendFrameDataBlocking(LPUART_Type *base, const uint8_t *txBuff, uint8_t txSize, uint32_t timeoutMSec)

Sends Frame data out through the LIN_LPUART module using blocking method. This function will calculate the checksum byte and send it with the frame data. Blocking means that the function does not return until the transmission is complete.

Parameters

- base – LPUART peripheral base address
- txBuff – source buffer containing 8-bit data chars to send
- txSize – the number of bytes to send
- timeoutMSec – timeout value in milli seconds

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_SendFrameData(LPUART_Type *base, const uint8_t *txBuff, uint8_t txSize)

Sends frame data out through the LIN_LPUART module using non-blocking method. This enables an a-sync method for transmitting data. Non-blocking means that the function returns immediately. The application has to get the transmit status to know when the transmit is complete. This function will calculate the checksum byte and send it with the frame data.

Parameters

- base – LPUART peripheral base address
- txBuff – source buffer containing 8-bit data chars to send

- txSize – the number of bytes to send

Returns
An error code or lin_status_t

lin_status_t LIN_LPUART_GetTransmitStatus(LPUART_Type *base, uint8_t *bytesRemaining)

Get status of an on-going non-blocking transmission While sending frame data using non-blocking method, users can use this function to get status of that transmission. This function return LIN_TX_BUSY while sending, or LIN_TIMEOUT if timeout has occurred, or return LIN_SUCCESS when the transmission is complete. The bytesRemaining shows number of bytes that still needed to transmit.

Parameters

- base – LPUART peripheral base address
- bytesRemaining – Number of bytes still needed to transmit

Returns
lin_status_t LIN_TX_BUSY, LIN_SUCCESS or LIN_TIMEOUT

lin_status_t LIN_LPUART_RecvFrmDataBlocking(LPUART_Type *base, uint8_t *rxBuff, uint8_t rxSize, uint32_t timeoutMSec)

Receives frame data through the LIN_LPUART module using blocking method. This function will check the checksum byte. If the checksum is correct, it will receive the frame data. Blocking means that the function does not return until the reception is complete.

Parameters

- base – LPUART peripheral base address
- rxBuff – buffer containing 8-bit received data
- rxSize – the number of bytes to receive
- timeoutMSec – timeout value in milli seconds

Returns
An error code or lin_status_t

lin_status_t LIN_LPUART_RecvFrmData(LPUART_Type *base, uint8_t *rxBuff, uint8_t rxSize)

Receives frame data through the LIN_LPUART module using non-blocking method. This function will check the checksum byte. If the checksum is correct, it will receive it with the frame data. Non-blocking means that the function returns immediately. The application has to get the receive status to know when the reception is complete.

Parameters

- base – LPUART peripheral base address
- rxBuff – buffer containing 8-bit received data
- rxSize – the number of bytes to receive

Returns
An error code or lin_status_t

lin_status_t LIN_LPUART_AbortTransferData(LPUART_Type *base)

Aborts an on-going non-blocking transmission/reception. While performing a non-blocking transferring data, users can call this function to terminate immediately the transferring.

Parameters

- base – LPUART peripheral base address

Returns
An error code or lin_status_t

lin_status_t LIN_LPUART_GetReceiveStatus(LPUART_Type *base, uint8_t *bytesRemaining)

Get status of an on-going non-blocking reception While receiving frame data using non-blocking method, users can use this function to get status of that receiving. This function return the current event ID, LIN_RX_BUSY while receiving and return LIN_SUCCESS, or time-out (LIN_TIMEOUT) when the reception is complete. The bytesRemaining shows number of bytes that still needed to receive.

Parameters

- base – LPUART peripheral base address
- bytesRemaining – Number of bytes still needed to receive

Returns

lin_status_t LIN_RX_BUSY, LIN_TIMEOUT or LIN_SUCCESS

lin_status_t LIN_LPUART_GoToSleepMode(LPUART_Type *base)

This function puts current node to sleep mode This function changes current node state to LIN_NODE_STATE_SLEEP_MODE.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_GotoIdleState(LPUART_Type *base)

Puts current LIN node to Idle state This function changes current node state to LIN_NODE_STATE_IDLE.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_SendWakeupSignal(LPUART_Type *base)

Sends a wakeup signal through the LIN_LPUART interface.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_MasterSendHeader(LPUART_Type *base, uint8_t id)

Sends frame header out through the LIN_LPUART module using a non-blocking method. This function sends LIN Break field, sync field then the ID with correct parity.

Parameters

- base – LPUART peripheral base address
- id – Frame Identifier

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_EnableIRQ(LPUART_Type *base)

Enables LIN_LPUART hardware interrupts.

Parameters

- base – LPUART peripheral base address

Returns
An error code or lin_status_t

lin_status_t LIN_LPUART_DisableIRQ(LPUART_Type *base)

Disables LIN_LPUART hardware interrupts.

Parameters

- base – LPUART peripheral base address

Returns
An error code or lin_status_t

lin_status_t LIN_LPUART_AutoBaudCapture(uint32_t instance)

This function capture bits time to detect break char, calculate baudrate from sync bits and enable transceiver if autobaud successful. This function should only be used in Slave. The timer should be in mode input capture of both rising and falling edges. The timer input capture pin should be externally connected to RXD pin.

Parameters

- instance – LPUART instance

Returns
lin_status_t

void LIN_LPUART_IRQHandler(LPUART_Type *base)

LIN_LPUART RX TX interrupt handler.

Parameters

- base – LPUART peripheral base address

LIN_LPUART_TRANSMISSION_COMPLETE_TIMEOUT

Max loops to wait for LPUART transmission complete.

When de-initializing the LIN LPUART module, the program shall wait for the previous transmission to complete. This parameter defines how many loops to check completion before return error. If defined as 0, driver will wait forever until completion.

AUTOBAUD_BAUDRATE_TOLERANCE

BIT_RATE_TOLERANCE_UNSYNC

BIT_DURATION_MAX_19200

BIT_DURATION_MIN_19200

BIT_DURATION_MAX_14400

BIT_DURATION_MIN_14400

BIT_DURATION_MAX_9600

BIT_DURATION_MIN_9600

BIT_DURATION_MAX_4800

BIT_DURATION_MIN_4800

BIT_DURATION_MAX_2400

BIT_DURATION_MIN_2400

TWO_BIT_DURATION_MAX_19200

TWO_BIT_DURATION_MIN_19200

TWO_BIT_DURATION_MAX_14400

TWO_BIT_DURATION_MIN_14400

TWO_BIT_DURATION_MAX_9600

TWO_BIT_DURATION_MIN_9600

TWO_BIT_DURATION_MAX_4800

TWO_BIT_DURATION_MIN_4800

TWO_BIT_DURATION_MAX_2400

TWO_BIT_DURATION_MIN_2400

AUTOBAUD_BREAK_TIME_MIN

2.33 LPADC: 12-bit SAR Analog-to-Digital Converter Driver

enum _lpadc_status_flags

Define hardware flags of the module.

Values:

enumerator kLPADC_ResultFIFO0OverflowFlag

Indicates that more data has been written to the Result FIFO 0 than it can hold.

enumerator kLPADC_ResultFIFO0ReadyFlag

Indicates when the number of valid datawords in the result FIFO 0 is greater than the setting watermark level.

enumerator kLPADC_TriggerExceptionFlag

Indicates that a trigger exception event has occurred.

enumerator kLPADC_TriggerCompletionFlag

Indicates that a trigger completion event has occurred.

enumerator kLPADC_CalibrationReadyFlag

Indicates that the calibration process is done.

enumerator kLPADC_ActiveFlag

Indicates that the ADC is in active state.

enumerator kLPADC_ResultFIFOOverflowFlag

To compilitable with old version, do not recommend using this, please use kLPADC_ResultFIFO0OverflowFlag as instead.

enumerator kLPADC_ResultFIFOReadyFlag

To compilitable with old version, do not recommend using this, please use kLPADC_ResultFIFO0ReadyFlag as instead.

enum _lpadc_interrupt_enable

Define interrupt switchers of the module.

Note: LPADC of different chips supports different number of trigger sources, please check the Reference Manual for details.

Values:

enumerator kLPADC_ResultFIFO0OverflowInterruptEnable
Configures ADC to generate overflow interrupt requests when FOF0 flag is asserted.

enumerator kLPADC_FIFO0WatermarkInterruptEnable
Configures ADC to generate watermark interrupt requests when RDY0 flag is asserted.

enumerator kLPADC_ResultFIFOOverflowInterruptEnable
To compilitable with old version, do not recommend using this, please use kLPADC_ResultFIFO0OverflowInterruptEnable as instead.

enumerator kLPADC_FIFOWatermarkInterruptEnable
To compilitable with old version, do not recommend using this, please use kLPADC_FIFO0WatermarkInterruptEnable as instead.

enumerator kLPADC_TriggerExceptionInterruptEnable
Configures ADC to generate trigger exception interrupt.

enumerator kLPADC_Trigger0CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 0 completion.

enumerator kLPADC_Trigger1CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 1 completion.

enumerator kLPADC_Trigger2CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 2 completion.

enumerator kLPADC_Trigger3CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 3 completion.

enumerator kLPADC_Trigger4CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 4 completion.

enumerator kLPADC_Trigger5CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 5 completion.

enumerator kLPADC_Trigger6CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 6 completion.

enumerator kLPADC_Trigger7CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 7 completion.

enumerator kLPADC_Trigger8CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 8 completion.

enumerator kLPADC_Trigger9CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 9 completion.

enumerator kLPADC_Trigger10CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 10 completion.

enumerator kLPADC_Trigger11CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 11 completion.

enumerator kLPADC_Trigger12CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 12 completion.

enumerator kLPADC_Trigger13CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 13 completion.

enumerator kLPADC_Trigger14CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 14 completion.

enumerator kLPADC_Trigger15CompletionInterruptEnable
Configures ADC to generate interrupt when trigger 15 completion.

enum _lpadc_trigger_status_flags
The enumerator of lpadc trigger status flags, including interrupted flags and completed flags.

Note: LPADC of different chips supports different number of trigger sources, please check the Reference Manual for details.

Values:

enumerator kLPADC_Trigger0InterruptedFlag
Trigger 0 is interrupted by a high priority exception.

enumerator kLPADC_Trigger1InterruptedFlag
Trigger 1 is interrupted by a high priority exception.

enumerator kLPADC_Trigger2InterruptedFlag
Trigger 2 is interrupted by a high priority exception.

enumerator kLPADC_Trigger3InterruptedFlag
Trigger 3 is interrupted by a high priority exception.

enumerator kLPADC_Trigger4InterruptedFlag
Trigger 4 is interrupted by a high priority exception.

enumerator kLPADC_Trigger5InterruptedFlag
Trigger 5 is interrupted by a high priority exception.

enumerator kLPADC_Trigger6InterruptedFlag
Trigger 6 is interrupted by a high priority exception.

enumerator kLPADC_Trigger7InterruptedFlag
Trigger 7 is interrupted by a high priority exception.

enumerator kLPADC_Trigger8InterruptedFlag
Trigger 8 is interrupted by a high priority exception.

enumerator kLPADC_Trigger9InterruptedFlag
Trigger 9 is interrupted by a high priority exception.

enumerator kLPADC_Trigger10InterruptedFlag
Trigger 10 is interrupted by a high priority exception.

enumerator kLPADC_Trigger11InterruptedFlag
Trigger 11 is interrupted by a high priority exception.

enumerator kLPADC_Trigger12InterruptedFlag
Trigger 12 is interrupted by a high priority exception.

enumerator kLPADC_Trigger13InterruptedFlag
Trigger 13 is interrupted by a high priority exception.

enumerator kLPADC_Trigger14InterruptedFlag
Trigger 14 is interrupted by a high priority exception.

enumerator kLPADC_Trigger15InterruptedFlag
Trigger 15 is interrupted by a high priority exception.

enumerator kLPADC_Trigger0CompletedFlag
Trigger 0 is completed and trigger 0 has enabled completion interrupts.

enumerator kLPADC_Trigger1CompletedFlag
Trigger 1 is completed and trigger 1 has enabled completion interrupts.

enumerator kLPADC_Trigger2CompletedFlag
Trigger 2 is completed and trigger 2 has enabled completion interrupts.

enumerator kLPADC_Trigger3CompletedFlag
Trigger 3 is completed and trigger 3 has enabled completion interrupts.

enumerator kLPADC_Trigger4CompletedFlag
Trigger 4 is completed and trigger 4 has enabled completion interrupts.

enumerator kLPADC_Trigger5CompletedFlag
Trigger 5 is completed and trigger 5 has enabled completion interrupts.

enumerator kLPADC_Trigger6CompletedFlag
Trigger 6 is completed and trigger 6 has enabled completion interrupts.

enumerator kLPADC_Trigger7CompletedFlag
Trigger 7 is completed and trigger 7 has enabled completion interrupts.

enumerator kLPADC_Trigger8CompletedFlag
Trigger 8 is completed and trigger 8 has enabled completion interrupts.

enumerator kLPADC_Trigger9CompletedFlag
Trigger 9 is completed and trigger 9 has enabled completion interrupts.

enumerator kLPADC_Trigger10CompletedFlag
Trigger 10 is completed and trigger 10 has enabled completion interrupts.

enumerator kLPADC_Trigger11CompletedFlag
Trigger 11 is completed and trigger 11 has enabled completion interrupts.

enumerator kLPADC_Trigger12CompletedFlag
Trigger 12 is completed and trigger 12 has enabled completion interrupts.

enumerator kLPADC_Trigger13CompletedFlag
Trigger 13 is completed and trigger 13 has enabled completion interrupts.

enumerator kLPADC_Trigger14CompletedFlag
Trigger 14 is completed and trigger 14 has enabled completion interrupts.

enumerator kLPADC_Trigger15CompletedFlag
Trigger 15 is completed and trigger 15 has enabled completion interrupts.

enum _lpadc_sample_scale_mode
Define enumeration of sample scale mode.

The sample scale mode is used to reduce the selected ADC analog channel input voltage level by a factor. The maximum possible voltage on the ADC channel input should be considered when selecting a scale mode to ensure that the reducing factor always results voltage level at or below the VREFH reference. This reducing capability allows conversion of analog inputs higher than VREFH. A-side and B-side channel inputs are both scaled using the scale mode.

Values:

enumerator kLPADC_SamplePartScale
Use divided input voltage signal. (For scale select,please refer to the reference manual).

enumerator kLPADC_SampleFullScale
Full scale (Factor of 1).

enum _lpadc_sample_channel_mode

Define enumeration of channel sample mode.

The channel sample mode configures the channel with single-end/differential/dual-single-end, side A/B.

Values:

enumerator kLPADC_SampleChannelSingleEndSideA

Single-end mode, only A-side channel is converted.

enumerator kLPADC_SampleChannelSingleEndSideB

Single-end mode, only B-side channel is converted.

enumerator kLPADC_SampleChannelDiffBothSideAB

Differential mode, the ADC result is (CHnA-CHnB).

enumerator kLPADC_SampleChannelDiffBothSideBA

Differential mode, the ADC result is (CHnB-CHnA).

enumerator kLPADC_SampleChannelDiffBothSide

Differential mode, the ADC result is (CHnA-CHnB).

enumerator kLPADC_SampleChannelDualSingleEndBothSide

Dual-Single-Ended Mode. Both A side and B side channels are converted independently.

enum _lpadc_hardware_average_mode

Define enumeration of hardware average selection.

It Selects how many ADC conversions are averaged to create the ADC result. An internal storage buffer is used to capture temporary results while the averaging iterations are executed.

Note: Some enumerator values are not available on some devices, mainly depends on the size of AVGS field in CMDH register.

Values:

enumerator kLPADC_HardwareAverageCount1

Single conversion.

enumerator kLPADC_HardwareAverageCount2

2 conversions averaged.

enumerator kLPADC_HardwareAverageCount4

4 conversions averaged.

enumerator kLPADC_HardwareAverageCount8

8 conversions averaged.

enumerator kLPADC_HardwareAverageCount16

16 conversions averaged.

enumerator kLPADC_HardwareAverageCount32

32 conversions averaged.

enumerator kLPADC_HardwareAverageCount64

64 conversions averaged.

enumerator kLPADC_HardwareAverageCount128

128 conversions averaged.

enum _lpadc_sample_time_mode

Define enumeration of sample time selection.

The shortest sample time maximizes conversion speed for lower impedance inputs. Extending sample time allows higher impedance inputs to be accurately sampled. Longer sample times can also be used to lower overall power consumption when command looping and sequencing is configured and high conversion rates are not required.

Values:

enumerator kLPADC_SampleTimeADCK3

3 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK5

5 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK7

7 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK11

11 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK19

19 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK35

35 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK67

69 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK131

131 ADCK cycles total sample time.

enum _lpadc_hardware_compare_mode

Define enumeration of hardware compare mode.

After an ADC channel input is sampled and converted and any averaging iterations are performed, this mode setting guides operation of the automatic compare function to optionally only store when the compare operation is true. When compare is enabled, the conversion result is compared to the compare values.

Values:

enumerator kLPADC_HardwareCompareDisabled

Compare disabled.

enumerator kLPADC_HardwareCompareStoreOnTrue

Compare enabled. Store on true.

enumerator kLPADC_HardwareCompareRepeatUntilTrue

Compare enabled. Repeat channel acquisition until true.

enum _lpadc_conversion_resolution_mode

Define enumeration of conversion resolution mode.

Configure the resolution bit in specific conversion type. For detailed resolution accuracy, see to lpadc_sample_channel_mode_t

Values:

enumerator kLPADC_ConversionResolutionStandard

Standard resolution. Single-ended 12-bit conversion, Differential 13-bit conversion with 2's complement output.

enumerator kLPADC_ConversionResolutionHigh
High resolution. Single-ended 16-bit conversion; Differential 16-bit conversion with 2's complement output.

enum _lpadc_conversion_average_mode
Define enumeration of conversion averages mode.

Configure the converion average number for auto-calibration.

Note: Some enumerator values are not available on some devices, mainly depends on the size of CAL_AVGS field in CTRL register.

Values:

enumerator kLPADC_ConversionAverage1
Single conversion.

enumerator kLPADC_ConversionAverage2
2 conversions averaged.

enumerator kLPADC_ConversionAverage4
4 conversions averaged.

enumerator kLPADC_ConversionAverage8
8 conversions averaged.

enumerator kLPADC_ConversionAverage16
16 conversions averaged.

enumerator kLPADC_ConversionAverage32
32 conversions averaged.

enumerator kLPADC_ConversionAverage64
64 conversions averaged.

enumerator kLPADC_ConversionAverage128
128 conversions averaged.

enumerator kLPADC_ConversionAverageMax

enum _lpadc_reference_voltage_mode
Define enumeration of reference voltage source.

For detail information, need to check the SoC's specification.

Values:

enumerator kLPADC_ReferenceVoltageAlt1
Option 1 setting.

enumerator kLPADC_ReferenceVoltageAlt2
Option 2 setting.

enumerator kLPADC_ReferenceVoltageAlt3
Option 3 setting.

enum _lpadc_power_level_mode
Define enumeration of power configuration.

Configures the ADC for power and performance. In the highest power setting the highest conversion rates will be possible. Refer to the device data sheet for power and performance capabilities for each setting.

Values:

enumerator kLPADC_PowerLevelAlt1
Lowest power setting.

enumerator kLPADC_PowerLevelAlt2
Next lowest power setting.

enumerator kLPADC_PowerLevelAlt3
...

enumerator kLPADC_PowerLevelAlt4
Highest power setting.

enum _lpadc_offset_calibration_mode
Define enumeration of offset calibration mode.

Values:

enumerator kLPADC_OffsetCalibration12bitMode
12 bit offset calibration mode.

enumerator kLPADC_OffsetCalibration16bitMode
16 bit offset calibration mode.

enum _lpadc_trigger_priority_policy
Define enumeration of trigger priority policy.

This selection controls how higher priority triggers are handled.

Note: kLPADC_TriggerPriorityPreemptSubsequently is not available on some devices, mainly depends on the size of TPRICTRL field in CFG register.

Values:

enumerator kLPADC_ConvPreemptImmediatelyNotAutoResumed
If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion is not automatically resumed or restarted.

enumerator kLPADC_ConvPreemptSoftlyNotAutoResumed
If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion is not resumed or restarted.

enumerator kLPADC_ConvPreemptImmediatelyAutoRestarted
If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion will automatically be restarted.

enumerator kLPADC_ConvPreemptSoftlyAutoRestarted
If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion will automatically be restarted.

enumerator kLPADC_ConvPreemptImmediatelyAutoResumed

If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion will automatically be resumed.

enumerator kLPADC_ConvPreemptSoftlyAutoResumed

If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion will be automatically be resumed.

enumerator kLPADC_TriggerPriorityPreemptImmediately

Legacy support is not recommended as it only ensures compatibility with older versions.

enumerator kLPADC_TriggerPriorityPreemptSoftly

Legacy support is not recommended as it only ensures compatibility with older versions.

enumerator kLPADC_TriggerPriorityExceptionDisabled

High priority trigger exception disabled.

enum _lpadc_tune_value

Define enumeration of tune value.

Values:

enumerator kLPADC_TuneValue0

Tune value 0.

enumerator kLPADC_TuneValue1

Tune value 1.

enumerator kLPADC_TuneValue2

Tune value 2.

enumerator kLPADC_TuneValue3

Tune value 3.

typedef enum *_lpadc_sample_scale_mode* lpadc_sample_scale_mode_t

Define enumeration of sample scale mode.

The sample scale mode is used to reduce the selected ADC analog channel input voltage level by a factor. The maximum possible voltage on the ADC channel input should be considered when selecting a scale mode to ensure that the reducing factor always results voltage level at or below the VREFH reference. This reducing capability allows conversion of analog inputs higher than VREFH. A-side and B-side channel inputs are both scaled using the scale mode.

typedef enum *_lpadc_sample_channel_mode* lpadc_sample_channel_mode_t

Define enumeration of channel sample mode.

The channel sample mode configures the channel with single-end/differential/dual-single-end, side A/B.

typedef enum *_lpadc_hardware_average_mode* lpadc_hardware_average_mode_t

Define enumeration of hardware average selection.

It Selects how many ADC conversions are averaged to create the ADC result. An internal storage buffer is used to capture temporary results while the averaging iterations are executed.

Note: Some enumerator values are not available on some devices, mainly depends on the size of AVGS field in CMDH register.

typedef enum *_lpadc_sample_time_mode* lpadc_sample_time_mode_t

Define enumeration of sample time selection.

The shortest sample time maximizes conversion speed for lower impedance inputs. Extending sample time allows higher impedance inputs to be accurately sampled. Longer sample times can also be used to lower overall power consumption when command looping and sequencing is configured and high conversion rates are not required.

typedef enum *_lpadc_hardware_compare_mode* lpadc_hardware_compare_mode_t

Define enumeration of hardware compare mode.

After an ADC channel input is sampled and converted and any averaging iterations are performed, this mode setting guides operation of the automatic compare function to optionally only store when the compare operation is true. When compare is enabled, the conversion result is compared to the compare values.

typedef enum *_lpadc_conversion_resolution_mode* lpadc_conversion_resolution_mode_t

Define enumeration of conversion resolution mode.

Configure the resolution bit in specific conversion type. For detailed resolution accuracy, see to lpadc_sample_channel_mode_t

typedef enum *_lpadc_conversion_average_mode* lpadc_conversion_average_mode_t

Define enumeration of conversion averages mode.

Configure the converion average number for auto-calibration.

Note: Some enumerator values are not available on some devices, mainly depends on the size of CAL_AVGS field in CTRL register.

typedef enum *_lpadc_reference_voltage_mode* lpadc_reference_voltage_source_t

Define enumeration of reference voltage source.

For detail information, need to check the SoC's specification.

typedef enum *_lpadc_power_level_mode* lpadc_power_level_mode_t

Define enumeration of power configuration.

Configures the ADC for power and performance. In the highest power setting the highest conversion rates will be possible. Refer to the device data sheet for power and performance capabilities for each setting.

typedef enum *_lpadc_offset_calibration_mode* lpadc_offset_calibration_mode_t

Define enumeration of offset calibration mode.

typedef enum *_lpadc_trigger_priority_policy* lpadc_trigger_priority_policy_t

Define enumeration of trigger priority policy.

This selection controls how higher priority triggers are handled.

Note: **kLPADC_TriggerPriorityPreemptSubsequently** is not available on some devices, mainly depends on the size of TPRICTRL field in CFG register.

typedef enum *_lpadc_tune_value* lpadc_tune_value_t

Define enumeration of tune value.

typedef struct *_lpadc_calibration_value* lpadc_calibration_value_t

A structure of calibration value.

LPADC_CONVERSION_COMPLETE_TIMEOUT

Max loops to wait for LPADC conversion complete.

When doing calibration, driver will wait for the completion of conversion. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

LPADC_CALIBRATION_READY_TIMEOUT

Max loops to wait for LPADC calibration ready.

Before doing calibration, driver will wait for the calibration ready. This parameter defines how many loops to check the calibration ready. If defined as 0, driver will wait forever until ready.

LPADC_GAIN_CAL_READY_TIMEOUT

Max loops to wait for LPADC gain calibration GAIN_CAL ready.

Before doing calibration, driver will wait for the gain calibration GAIN_CAL ready. This parameter defines how many loops to check the gain calibration GAIN_CAL ready. If defined as 0, driver will wait forever until ready.

ADC_OFSTRIM_OFSTRIM_MAX

ADC_OFSTRIM_OFSTRIM_SIGN

LPADC_GET_ACTIVE_COMMAND_STATUS(statusVal)

Define the MACRO function to get command status from status value.

The statusVal is the return value from LPADC_GetStatusFlags().

LPADC_GET_ACTIVE_TRIGGER_STATUE(statusVal)

Define the MACRO function to get trigger status from status value.

The statusVal is the return value from LPADC_GetStatusFlags().

void LPADC_Init(ADC_Type *base, const *lpadc_config_t* *config)

Initializes the LPADC module.

Parameters

- base – LPADC peripheral base address.
- config – Pointer to configuration structure. See "lpadc_config_t".

void LPADC_GetDefaultConfig(*lpadc_config_t* *config)

Gets an available pre-defined settings for initial configuration.

This function initializes the converter configuration structure with an available settings. The default values are:

```
config->enableInDozeMode        = true;
config->enableAnalogPreliminary = false;
config->powerUpDelay            = 0x80;
config->referenceVoltageSource  = kLPADC_ReferenceVoltageAlt1;
config->powerLevelMode          = kLPADC_PowerLevelAlt1;
config->triggerPriorityPolicy   = kLPADC_TriggerPriorityPreemptImmediately;
config->enableConvPause         = false;
config->convPauseDelay          = 0U;
config->FIFOWatermark           = 0U;
```

Parameters

- config – Pointer to configuration structure.

void LPADC_Deinit(ADC_Type *base)

De-initializes the LPADC module.

Parameters

- base – LPADC peripheral base address.

static inline void LPADC_Enable(ADC_Type *base, bool enable)

Switch on/off the LPADC module.

Parameters

- base – LPADC peripheral base address.
- enable – switcher to the module.

static inline void LPADC_DoResetFIFO(ADC_Type *base)

Do reset the conversion FIFO.

Parameters

- base – LPADC peripheral base address.

static inline void LPADC_DoResetConfig(ADC_Type *base)

Do reset the module's configuration.

Reset all ADC internal logic and registers, except the Control Register (ADCx_CTRL).

Parameters

- base – LPADC peripheral base address.

static inline uint32_t LPADC_GetStatusFlags(ADC_Type *base)

Get status flags.

Parameters

- base – LPADC peripheral base address.

Returns

status flags' mask. See to _lpadc_status_flags.

static inline void LPADC_ClearStatusFlags(ADC_Type *base, uint32_t mask)

Clear status flags.

Only the flags can be cleared by writing ADCx_STATUS register would be cleared by this API.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for flags to be cleared. See to _lpadc_status_flags.

static inline uint32_t LPADC_GetTriggerStatusFlags(ADC_Type *base)

Get trigger status flags to indicate which trigger sequences have been completed or interrupted by a high priority trigger exception.

Parameters

- base – LPADC peripheral base address.

Returns

The OR'ed value of _lpadc_trigger_status_flags.

static inline void LPADC_ClearTriggerStatusFlags(ADC_Type *base, uint32_t mask)

Clear trigger status flags.

Parameters

- base – LPADC peripheral base address.

- mask – The mask of trigger status flags to be cleared, should be the OR'ed value of _lpadc_trigger_status_flags.

static inline void LPADC_EnableInterrupts(ADC_Type *base, uint32_t mask)

Enable interrupts.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for interrupt events. See to _lpadc_interrupt_enable.

static inline void LPADC_DisableInterrupts(ADC_Type *base, uint32_t mask)

Disable interrupts.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for interrupt events. See to _lpadc_interrupt_enable.

static inline void LPADC_EnableFIFOWatermarkDMA(ADC_Type *base, bool enable)

Switch on/off the DMA trigger for FIFO watermark event.

Parameters

- base – LPADC peripheral base address.
- enable – Switcher to the event.

static inline uint32_t LPADC_GetConvResultCount(ADC_Type *base)

Get the count of result kept in conversion FIFO.

Parameters

- base – LPADC peripheral base address.

Returns

The count of result kept in conversion FIFO.

bool LPADC_GetConvResult(ADC_Type *base, *lpadc_conv_result_t* *result)

Get the result in conversion FIFO.

Parameters

- base – LPADC peripheral base address.
- result – Pointer to structure variable that keeps the conversion result in conversion FIFO.

Returns

Status whether FIFO entry is valid.

void LPADC_GetConvResultBlocking(ADC_Type *base, *lpadc_conv_result_t* *result)

Get the result in conversion FIFO using blocking method.

Parameters

- base – LPADC peripheral base address.
- result – Pointer to structure variable that keeps the conversion result in conversion FIFO.

void LPADC_SetConvTriggerConfig(ADC_Type *base, uint32_t triggerId, const *lpadc_conv_trigger_config_t* *config)

Configure the conversion trigger source.

Each programmable trigger can launch the conversion command in command buffer.

Parameters

- base – LPADC peripheral base address.
- triggerId – ID for each trigger. Typically, the available value range is from 0.
- config – Pointer to configuration structure. See to lpadc_conv_trigger_config_t.

void LPADC_GetDefaultConvTriggerConfig(*lpadc_conv_trigger_config_t* *config)

Gets an available pre-defined settings for trigger's configuration.

This function initializes the trigger's configuration structure with an available settings. The default values are:

```
config->targetCommandId       = 0U;
config->delayPower            = 0U;
config->priority              = 0U;
config->channelAFIFOSelect    = 0U;
config->channelBFIFOSelect    = 0U;
config->enableHardwareTrigger = false;
```

Parameters

- config – Pointer to configuration structure.

static inline void LPADC_DoSoftwareTrigger(ADC_Type *base, uint32_t triggerIdMask)

Do software trigger to conversion command.

Parameters

- base – LPADC peripheral base address.
- triggerIdMask – Mask value for software trigger indexes, which count from zero.

static inline void LPADC_EnableHardwareTriggerCommandSelection(ADC_Type *base, uint32_t triggerId, bool enable)

Enable hardware trigger command selection.

This function will use the hardware trigger command from ADC_ETC.The trigger command is then defined by ADC hardware trigger command selection field in ADC_ETC->TRIGx_CHAINy_z_n[CSEL].

Parameters

- base – LPADC peripheral base address.
- triggerId – ID for each trigger. Typically, the available value range is from 0.
- enable – True to enable or flase to disable.

void LPADC_SetConvCommandConfig(ADC_Type *base, uint32_t commandId, const *lpadc_conv_command_config_t* *config)

Configure conversion command.

Note: The number of compare value register on different chips is different, that is mean in some chips, some command buffers do not have the compare functionality.

Parameters

- base – LPADC peripheral base address.
- commandId – ID for command in command buffer. Typically, the available value range is 1 - 15.

- config – Pointer to configuration structure. See to lpadc_conv_command_config_t.

void LPADC_GetDefaultConvCommandConfig(*lpadc_conv_command_config_t* *config)

Gets an available pre-defined settings for conversion command's configuration.

This function initializes the conversion command's configuration structure with an available settings. The default values are:

```
config->sampleScaleMode            = kLPADC_SampleFullScale;
config->channelBScaleMode          = kLPADC_SampleFullScale;
config->sampleChannelMode          = kLPADC_SampleChannelSingleEndSideA;
config->channelNumber              = 0U;
config->channelBNumber             = 0U;
config->chainedNextCommandNumber   = 0U;
config->enableAutoChannelIncrement = false;
config->loopCount                  = 0U;
config->hardwareAverageMode        = kLPADC_HardwareAverageCount1;
config->sampleTimeMode             = kLPADC_SampleTimeADCK3;
config->hardwareCompareMode        = kLPADC_HardwareCompareDisabled;
config->hardwareCompareValueHigh   = 0U;
config->hardwareCompareValueLow    = 0U;
config->conversionResolutionMode   = kLPADC_ConversionResolutionStandard;
config->enableWaitTrigger          = false;
config->enableChannelB             = false;
```

Parameters

- config – Pointer to configuration structure.

void LPADC_EnableCalibration(ADC_Type *base, bool enable)

Enable the calibration function.

When CALOFS is set, the ADC is configured to perform a calibration function anytime the ADC executes a conversion. Any channel selected is ignored and the value returned in the RESFIFO is a signed value between -31 and 31. -32 is not a valid and is never a returned value. Software should copy the lower 6- bits of the conversion result stored in the RESFIFO after a completed calibration conversion to the OFSTRIM field. The OFSTRIM field is used in normal operation for offset correction.

Parameters

- base – LPADC peripheral base address.
- enable – switcher to the calibration function.

static inline void LPADC_SetOffsetValue(ADC_Type *base, uint32_t value)

Set proper offset value to trim ADC.

To minimize the offset during normal operation, software should read the conversion result from the RESFIFO calibration operation and write the lower 6 bits to the OFSTRIM register.

Parameters

- base – LPADC peripheral base address.
- value – Setting offset value.

status_t LPADC_DoAutoCalibration(ADC_Type *base)

Do auto calibration.

Calibration function should be executed before using converter in application. It used the software trigger and a dummy conversion, get the offset and write them into the OFSTRIM register. It called some of functional API including:

- LPADC_EnableCalibration(...)

- LPADC_SetOffsetValue(...)
- LPADC_SetConvCommandConfig(...)
- LPADC_SetConvTriggerConfig(...)

Parameters

- base – LPADC peripheral base address.
- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

static inline void LPADC_SetOffsetValue(ADC_Type *base, int16_t value)

Set trim value for offset.

Note: For 16-bit conversions, each increment is 1/2 LSB resulting in a programmable offset range of -256 LSB to 255.5 LSB; For 12-bit conversions, each increment is 1/32 LSB resulting in a programmable offset range of -16 LSB to 15.96875 LSB.

Parameters

- base – LPADC peripheral base address.
- value – Offset trim value, is a 10-bit signed value between -512 and 511.

static inline void LPADC_GetOffsetValue(ADC_Type *base, int16_t *pValue)

Get trim value of offset.

Parameters

- base – LPADC peripheral base address.
- pValue – Pointer to the variable in type of int16_t to store offset value.

static inline void LPADC_EnableOffsetCalibration(ADC_Type *base, bool enable)

Enable the offset calibration function.

Parameters

- base – LPADC peripheral base address.
- enable – switcher to the calibration function.

static inline void LPADC_SetOffsetCalibrationMode(ADC_Type *base, *lpadc_offset_calibration_mode_t* mode)

Set offset calibration mode.

Parameters

- base – LPADC peripheral base address.
- mode – set offset calibration mode.see to lpadc_offset_calibration_mode_t .

status_t LPADC_DoOffsetCalibration(ADC_Type *base)

Do offset calibration.

Parameters

- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.

- kStatus_Timeout – Timeout occurs while waiting completion.

void LPADC_PrepareAutoCalibration(ADC_Type *base)

Prepare auto calibration, LPADC_FinishAutoCalibration has to be called before using the LPADC. LPADC_DoAutoCalibration has been split in two API to avoid to be stuck too long in the function.

Parameters

- base – LPADC peripheral base address.

status_t LPADC_FinishAutoCalibration(ADC_Type *base)

Finish auto calibration start with LPADC_PrepareAutoCalibration.

Note: This feature is used for LPADC with CTRL[CALOFSMODE].

Parameters

- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

void LPADC_GetCalibrationValue(ADC_Type *base, *lpadc_calibration_value_t* *ptrCalibrationValue)

Get calibration value into the memory which is defined by invoker.

Note: Please note the ADC will be disabled temporary.

Note: This function should be used after finish calibration.

Parameters

- base – LPADC peripheral base address.
- ptrCalibrationValue – Pointer to lpadc_calibration_value_t structure, this memory block should be always powered on even in low power modes.

status_t LPADC_SetCalibrationValue(ADC_Type *base, const *lpadc_calibration_value_t* *ptrCalibrationValue)

Set calibration value into ADC calibration registers.

Note: Please note the ADC will be disabled temporary.

Parameters

- base – LPADC peripheral base address.
- ptrCalibrationValue – Pointer to lpadc_calibration_value_t structure which contains ADC's calibration value.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

static inline void LPADC_RequestHighSpeedModeTrim(ADC_Type *base)

Request high speed mode trim calculation.

Parameters

- base – LPADC peripheral base address.

static inline int8_t LPADC_GetHighSpeedTrimValue(ADC_Type *base)

Get high speed mode trim value, the result is a 5-bit signed value between -16 and 15.

Note: The high speed mode trim value is used to minimize offset for high speed conversion.

Parameters

- base – LPADC peripheral base address.

Returns

The calculated high speed mode trim value.

static inline void LPADC_SetHighSpeedTrimValue(ADC_Type *base, int8_t trimValue)

Set high speed mode trim value.

Note: If is possible to set the trim value manually, but it is recommended to use the LPADC_RequestHighSpeedModeTrim.

Parameters

- base – LPADC peripheral base address.
- trimValue – The trim value to be set.

static inline void LPADC_EnableHighSpeedConversionMode(ADC_Type *base, bool enable)

Enable/disable high speed conversion mode, if enabled conversions complete 2 or 3 ADCK cycles sooner compared to conversion cycle counts when high speed mode is disabled.

Parameters

- base – LPADC peripheral base address.
- enable – Used to enable/disable high speed conversion mode:
  - **true** Enable high speed conversion mode;
  - **false** Disable high speed conversion mode.

static inline void LPADC_EnableExtraCycle(ADC_Type *base, bool enable)

Enable/disable an additional ADCK cycle to conversion.

Parameters

- base – LPADC peripheral base address.
- enable – Used to enable/disable an additional ADCK cycle to conversion:
  - **true** Enable an additional ADCK cycle to conversion;
  - **false** Disable an additional ADCK cycle to conversion.

static inline void LPADC_SetTuneValue(ADC_Type *base, *lpadc_tune_value_t* tuneValue)

Set tune value which provides some variability in how many cycles are needed to complete a conversion.

Parameters

- base – LPADC peripheral base address.

- tuneValue – The tune value to be set, please refer to lpadc_tune_value_t.

static inline *lpadc_tune_value_t* LPADC_GetTuneValue(ADC_Type *base)

Get tune value which provides some variability in how many cycles are needed to complete a conversion.

Parameters

- base – LPADC peripheral base address.

Returns

The tune value, please refer to lpadc_tune_value_t.

FSL_LPADC_DRIVER_VERSION

LPADC driver version 2.9.5.

struct lpadc_config_t

#include <fsl_lpadc.h> LPADC global configuration.

This structure would used to keep the settings for initialization.

Public Members

bool enableInternalClock

Enables the internally generated clock source. The clock source is used in clock selection logic at the chip level and is optionally used for the ADC clock source.

bool enableVref1LowVoltage

If voltage reference option1 input is below 1.8V, it should be "true". If voltage reference option1 input is above 1.8V, it should be "false".

bool enableInDozeMode

Control system transition to Stop and Wait power modes while ADC is converting. When enabled in Doze mode, immediate entries to Wait or Stop are allowed. When disabled, the ADC will wait for the current averaging iteration/FIFO storage to complete before acknowledging stop or wait mode entry.

lpadc_conversion_average_mode_t conversionAverageMode

Auto-Calibration Averages.

bool enableAnalogPreliminary

ADC analog circuits are pre-enabled and ready to execute conversions without startup delays(at the cost of higher DC current consumption).

uint32_t powerUpDelay

When the analog circuits are not pre-enabled, the ADC analog circuits are only powered while the ADC is active and there is a counted delay defined by this field after an initial trigger transitions the ADC from its Idle state to allow time for the analog circuits to stabilize. The startup delay count of (powerUpDelay * 4) ADCK cycles must result in a longer delay than the analog startup time.

lpadc_reference_voltage_source_t referenceVoltageSource

Selects the voltage reference high used for conversions.

lpadc_power_level_mode_t powerLevelMode

Power Configuration Selection.

lpadc_trigger_priority_policy_t triggerPriorityPolicy

Control how higher priority triggers are handled, see to lpadc_trigger_priority_policy_t.

bool enableConvPause

Enables the ADC pausing function. When enabled, a programmable delay is inserted during command execution sequencing between LOOP iterations, between commands in a sequence, and between conversions when command is executing in "Compare Until True" configuration.

uint32_t convPauseDelay

Controls the duration of pausing during command execution sequencing. The pause delay is a count of (convPauseDelay*4) ADCK cycles. Only available when ADC pausing function is enabled. The available value range is in 9-bit.

uint32_t FIFOWatermark

FIFOWatermark is a programmable threshold setting. When the number of datawords stored in the ADC Result FIFO is greater than the value in this field, the ready flag would be asserted to indicate stored data has reached the programmable threshold.

struct lpadc_conv_command_config_t

#include <fsl_lpadc.h> Define structure to keep the configuration for conversion command.

Public Members

lpadc_sample_scale_mode_t sampleScaleMode

Sample scale mode.

lpadc_sample_scale_mode_t channelBScaleMode

Alternate channe B Scale mode.

lpadc_sample_channel_mode_t sampleChannelMode

Channel sample mode.

uint32_t channelNumber

Channel number, select the channel or channel pair.

uint32_t channelBNumber

Alternate Channel B number, select the channel.

uint32_t chainedNextCommandNumber

Selects the next command to be executed after this command completes. 1-15 is available, 0 is to terminate the chain after this command.

bool enableAutoChannelIncrement

Loop with increment: when disabled, the "loopCount" field selects the number of times the selected channel is converted consecutively; when enabled, the "loopCount" field defines how many consecutive channels are converted as part of the command execution.

uint32_t loopCount

Selects how many times this command executes before finish and transition to the next command or Idle state. Command executes LOOP+1 times. 0-15 is available.

lpadc_hardware_average_mode_t hardwareAverageMode

Hardware average selection.

lpadc_sample_time_mode_t sampleTimeMode

Sample time selection.

lpadc_hardware_compare_mode_t hardwareCompareMode

Hardware compare selection.

uint32_t hardwareCompareValueHigh

Compare Value High. The available value range is in 16-bit.

uint32_t hardwareCompareValueLow

Compare Value Low. The available value range is in 16-bit.

lpadc_conversion_resolution_mode_t conversionResolutionMode

Conversion resolution mode.

bool enableWaitTrigger

Wait for trigger assertion before execution: when disabled, this command will be automatically executed; when enabled, the active trigger must be asserted again before executing this command.

struct lpadc_conv_trigger_config_t

#include <fsl_lpadc.h> Define structure to keep the configuration for conversion trigger.

Public Members

uint32_t targetCommandId

Select the command from command buffer to execute upon detect of the associated trigger event.

uint32_t delayPower

Select the trigger delay duration to wait at the start of servicing a trigger event. When this field is clear, then no delay is incurred. When this field is set to a non-zero value, the duration for the delay is 2^delayPower ADCK cycles. The available value range is 4-bit.

uint32_t priority

Sets the priority of the associated trigger source. If two or more triggers have the same priority level setting, the lower order trigger event has the higher priority. The lower value for this field is for the higher priority, the available value range is 1-bit.

bool enableHardwareTrigger

Enable hardware trigger source to initiate conversion on the rising edge of the input trigger source or not. THe software trigger is always available.

struct lpadc_conv_result_t

#include <fsl_lpadc.h> Define the structure to keep the conversion result.

Public Members

uint32_t commandIdSource

Indicate the command buffer being executed that generated this result.

uint32_t loopCountIndex

Indicate the loop count value during command execution that generated this result.

uint32_t triggerIdSource

Indicate the trigger source that initiated a conversion and generated this result.

uint16_t convValue

Data result.

struct _lpadc_calibration_value

#include <fsl_lpadc.h> A structure of calibration value.

2.34 LPCMP: Low Power Analog Comparator Driver

void LPCMP_Init(LPCMP_Type *base, const *lpcmp_config_t* *config)

Initialize the LPCMP.

This function initializes the LPCMP module. The operations included are:

- Enabling the clock for LPCMP module.
- Configuring the comparator.
- Enabling the LPCMP module. Note: For some devices, multiple LPCMP instance share the same clock gate. In this case, to enable the clock for any instance enables all the LPCMPs. Check the chip reference manual for the clock assignment of the LPCMP.

Parameters

- base – LPCMP peripheral base address.
- config – Pointer to "lpcmp_config_t" structure.

void LPCMP_Deinit(LPCMP_Type *base)

De-initializes the LPCMP module.

This function de-initializes the LPCMP module. The operations included are:

- Disabling the LPCMP module.
- Disabling the clock for LPCMP module.

This function disables the clock for the LPCMP. Note: For some devices, multiple LPCMP instance shares the same clock gate. In this case, before disabling the clock for the LPCMP, ensure that all the LPCMP instances are not used.

Parameters

- base – LPCMP peripheral base address.

void LPCMP_GetDefaultConfig(*lpcmp_config_t* *config)

Gets an available pre-defined settings for the comparator's configuration.

This function initializes the comparator configuration structure to these default values:

```
config->enableStopMode       = false;
config->enableOutputPin      = false;
config->enableCmpToDacLink   = false;
config->useUnfilteredOutput  = false;
config->enableInvertOutput   = false;
config->hysteresisMode       = kLPCMP_HysteresisLevel0;
config->powerMode            = kLPCMP_LowSpeedPowerMode;
config->functionalSourceClock = kLPCMP_FunctionalClockSource0;
config->plusInputSrc         = kLPCMP_PlusInputSrcMux;
config->minusInputSrc        = kLPCMP_MinusInputSrcMux;
```

Parameters

- config – Pointer to "lpcmp_config_t" structure.

static inline void LPCMP_Enable(LPCMP_Type *base, bool enable)

Enable/Disable LPCMP module.

Parameters

- base – LPCMP peripheral base address.
- enable – "true" means enable the module, and "false" means disable the module.

void LPCMP_SetInputChannels(LPCMP_Type *base, uint32_t positiveChannel, uint32_t negativeChannel)

Select the input channels for LPCMP. This function determines which input is selected for the negative and positive mux.

Parameters

- base – LPCMP peripheral base address.
- positiveChannel – Positive side input channel number. Available range is 0-7.
- negativeChannel – Negative side input channel number. Available range is 0-7.

static inline void LPCMP_EnableDMA(LPCMP_Type *base, bool enable)

Enables/disables the DMA request for rising/falling events. Normally, the LPCMP generates a CPU interrupt if there is a rising/falling event. When DMA support is enabled and the rising/falling interrupt is enabled , the rising/falling event forces a DMA transfer request rather than a CPU interrupt instead.

Parameters

- base – LPCMP peripheral base address.
- enable – "true" means enable DMA support, and "false" means disable DMA support.

void LPCMP_SetFilterConfig(LPCMP_Type *base, const *lpcmp_filter_config_t* *config)

Configures the filter.

Parameters

- base – LPCMP peripheral base address.
- config – Pointer to "lpcmp_filter_config_t" structure.

void LPCMP_SetDACConfig(LPCMP_Type *base, const *lpcmp_dac_config_t* *config)

Configure the internal DAC module.

Parameters

- base – LPCMP peripheral base address.
- config – Pointer to "lpcmp_dac_config_t" structure. If config is "NULL", disable internal DAC.

static inline void LPCMP_EnableInterrupts(LPCMP_Type *base, uint32_t mask)

Enable the interrupts.

Parameters

- base – LPCMP peripheral base address.
- mask – Mask value for interrupts. See "_lpcmp_interrupt_enable".

static inline void LPCMP_DisableInterrupts(LPCMP_Type *base, uint32_t mask)

Disable the interrupts.

Parameters

- base – LPCMP peripheral base address.
- mask – Mask value for interrupts. See "_lpcmp_interrupt_enable".

static inline uint32_t LPCMP_GetStatusFlags(LPCMP_Type *base)

Get the LPCMP status flags.

Parameters

- base – LPCMP peripheral base address.

Returns
Mask value for the asserted flags. See "_lpcmp_status_flags".

static inline void LPCMP_ClearStatusFlags(LPCMP_Type *base, uint32_t mask)

Clear the LPCMP status flags.

Parameters

- base – LPCMP peripheral base address.
- mask – Mask value for the flags. See "_lpcmp_status_flags".

static inline void LPCMP_EnableWindowMode(LPCMP_Type *base, bool enable)

Enable/Disable window mode.When any windowed mode is active, COUTA is clocked by the bus clock whenever WINDOW = 1. The last latched value is held when WINDOW = 0. The optionally inverted comparator output COUT_RAW is sampled on every bus clock when WINDOW=1 to generate COUTA.

Parameters

- base – LPCMP peripheral base address.
- enable – "true" means enable window mode, and "false" means disable window mode.

void LPCMP_SetWindowControl(LPCMP_Type *base, const *lpcmp_window_control_config_t* *config)

Configure the window control, users can use this API to implement operations on the window, such as inverting the window signal, setting the window closing event(only valid in windowing mode), and setting the COUTA signal after the window is closed(only valid in windowing mode).

Parameters

- base – LPCMP peripheral base address.
- config – Pointer "lpcmp_window_control_config_t" structure.

FSL_LPCMP_DRIVER_VERSION

LPCMP driver version 2.3.2.

enum _lpcmp_status_flags

LPCMP status falgs mask.

Values:

enumerator kLPCMP_OutputRisingEventFlag

Rising-edge on the comparison output has occurred.

enumerator kLPCMP_OutputFallingEventFlag

Falling-edge on the comparison output has occurred.

enumerator kLPCMP_OutputAssertEventFlag

Return the current value of the analog comparator output. The flag does not support W1C.

enum _lpcmp_interrupt_enable

LPCMP interrupt enable/disable mask.

Values:

enumerator kLPCMP_OutputRisingInterruptEnable

Comparator interrupt enable rising.

enumerator kLPCMP_OutputFallingInterruptEnable
Comparator interrupt enable falling.

enum _lpcmp_hysteresis_mode
LPCMP hysteresis mode. See chip data sheet to get the actual hystersis value with each level.

Values:

enumerator kLPCMP_HysteresisLevel0
The hard block output has level 0 hysteresis internally.

enumerator kLPCMP_HysteresisLevel1
The hard block output has level 1 hysteresis internally.

enumerator kLPCMP_HysteresisLevel2
The hard block output has level 2 hysteresis internally.

enumerator kLPCMP_HysteresisLevel3
The hard block output has level 3 hysteresis internally.

enum _lpcmp_power_mode
LPCMP nano mode.

Values:

enumerator kLPCMP_LowSpeedPowerMode
Low speed comparison mode is selected.

enumerator kLPCMP_HighSpeedPowerMode
High speed comparison mode is selected.

enumerator kLPCMP_NanoPowerMode
Nano power comparator is enabled.

enum _lpcmp_dac_reference_voltage_source
Internal DAC reference voltage source.

Values:

enumerator kLPCMP_VrefSourceVin1
vrefh_int is selected as resistor ladder network supply reference Vin.

enumerator kLPCMP_VrefSourceVin2
vrefh_ext is selected as resistor ladder network supply reference Vin.

enum _lpcmp_couta_signal
Set the COUTA signal value when the window is closed.

Values:

enumerator kLPCMP_COUTASignalNoSet
NO set the COUTA signal value when the window is closed.

enumerator kLPCMP_COUTASignalLow
Set COUTA signal low(0) when the window is closed.

enumerator kLPCMP_COUTASignalHigh
Set COUTA signal high(1) when the window is closed.

enum _lpcmp_close_window_event
Set COUT event, which can close the active window in window mode.

Values:

enumerator kLPCMP_CLoseWindowEventNoSet
No Set COUT event, which can close the active window in window mode.

enumerator kLPCMP_CloseWindowEventRisingEdge
Set rising edge COUT signal as COUT event.

enumerator kLPCMP_CloseWindowEventFallingEdge
Set falling edge COUT signal as COUT event.

enumerator kLPCMP_CLoseWindowEventBothEdge
Set both rising and falling edge COUT signal as COUT event.

typedef enum *_lpcmp_hysteresis_mode* lpcmp_hysteresis_mode_t
LPCMP hysteresis mode. See chip data sheet to get the actual hystersis value with each level.

typedef enum *_lpcmp_power_mode* lpcmp_power_mode_t
LPCMP nano mode.

typedef enum *_lpcmp_dac_reference_voltage_source* lpcmp_dac_reference_voltage_source_t
Internal DAC reference voltage source.

typedef enum *_lpcmp_couta_signal* lpcmp_couta_signal_t
Set the COUTA signal value when the window is closed.

typedef enum *_lpcmp_close_window_event* lpcmp_close_window_event_t
Set COUT event, which can close the active window in window mode.

typedef struct *_lpcmp_filter_config* lpcmp_filter_config_t
Configure the filter.

typedef struct *_lpcmp_dac_config* lpcmp_dac_config_t
configure the internal DAC.

typedef struct *_lpcmp_config* lpcmp_config_t
Configures the comparator.

typedef struct *_lpcmp_window_control_config* lpcmp_window_control_config_t
Configure the window mode control.

LPCMP_CCR1_COUTA_CFG_MASK

LPCMP_CCR1_COUTA_CFG_SHIFT

LPCMP_CCR1_COUTA_CFG(x)

LPCMP_CCR1_EVT_SEL_CFG_MASK

LPCMP_CCR1_EVT_SEL_CFG_SHIFT

LPCMP_CCR1_EVT_SEL_CFG(x)

struct _lpcmp_filter_config
#include <fsl_lpcmp.h> Configure the filter.

Public Members

bool enableSample
Decide whether to use the external SAMPLE as a sampling clock input.

uint8_t filterSampleCount
Filter Sample Count. Available range is 1-7; 0 disables the filter.

uint8_t filterSamplePeriod

Filter Sample Period. The divider to the bus clock. Available range is 0-255. The sampling clock must be at least 4 times slower than the system clock to the comparator. So if enableSample is "false", filterSamplePeriod should be set greater than 4.

struct __lpcmp__dac__config

#include <fsl_lpcmp.h> configure the internal DAC.

Public Members

bool enableLowPowerMode

Decide whether to enable DAC low power mode.

lpcmp_dac_reference_voltage_source_t referenceVoltageSource

Internal DAC supply voltage reference source.

uint8_t DACValue

Value for the DAC Output Voltage. Different devices has different available range, for specific values, please refer to the reference manual.

struct __lpcmp__config

#include <fsl_lpcmp.h> Configures the comparator.

Public Members

bool enableOutputPin

Decide whether to enable the comparator is available in selected pin.

bool useUnfilteredOutput

Decide whether to use unfiltered output.

bool enableInvertOutput

Decide whether to inverts the comparator output.

lpcmp_hysteresis_mode_t hysteresisMode

LPCMP hysteresis mode.

lpcmp_power_mode_t powerMode

LPCMP power mode.

struct __lpcmp__window__control__config

#include <fsl_lpcmp.h> Configure the window mode control.

Public Members

bool enableInvertWindowSignal

True: enable invert window signal, False: disable invert window signal.

lpcmp_couta_signal_t COUTASignal

Decide whether to define the COUTA signal value when the window is closed.

lpcmp_close_window_event_t closeWindowEvent

Decide whether to select COUT event signal edge defines a COUT event to close window.

2.35 LPI2C: Low Power Inter-Integrated Circuit Driver

void LPI2C_DriverIRQHandler(uint32_t instance)

LPI2C driver IRQ handler common entry.

This function provides the common IRQ request entry for LPI2C.

Parameters

- instance – LPI2C instance.

FSL_LPI2C_DRIVER_VERSION

LPI2C driver version.

LPI2C status return codes.

Values:

enumerator kStatus_LPI2C_Busy

The master is already performing a transfer.

enumerator kStatus_LPI2C_Idle

The slave driver is idle.

enumerator kStatus_LPI2C_Nak

The slave device sent a NAK in response to a byte.

enumerator kStatus_LPI2C_FifoError

FIFO under run or overrun.

enumerator kStatus_LPI2C_BitError

Transferred bit was not seen on the bus.

enumerator kStatus_LPI2C_ArbitrationLost

Arbitration lost error.

enumerator kStatus_LPI2C_PinLowTimeout

SCL or SDA were held low longer than the timeout.

enumerator kStatus_LPI2C_NoTransferInProgress

Attempt to abort a transfer when one is not in progress.

enumerator kStatus_LPI2C_DmaRequestFail

DMA request failed.

enumerator kStatus_LPI2C_Timeout

Timeout polling status flags.

IRQn_Type const kLpi2cMasterIrqs[]

Array to map LPI2C instance number to IRQ number, used internally for LPI2C master interrupt and EDMA transactional APIs.

IRQn_Type const kLpi2cSlaveIrqs[]

lpi2c_master_isr_t s_lpi2cMasterIsr

Pointer to master IRQ handler for each instance, used internally for LPI2C master interrupt and EDMA transactional APIs.

void *s_lpi2cMasterHandle[]

Pointers to master handles for each instance, used internally for LPI2C master interrupt and EDMA transactional APIs.

uint32_t LPI2C_GetInstance(LPI2C_Type *base)

Returns an instance number given a base address.

If an invalid base address is passed, debug builds will assert. Release builds will just return instance number 0.

Parameters

- base – The LPI2C peripheral base address.

Returns

LPI2C instance number starting from 0.

I2C_RETRY_TIMES

Retry times for waiting flag.

2.36 LPI2C Master Driver

void LPI2C_MasterGetDefaultConfig(*lpi2c_master_config_t* *masterConfig)

Provides a default configuration for the LPI2C master peripheral.

This function provides the following default configuration for the LPI2C master peripheral:

```
masterConfig->enableMaster            = true;
masterConfig->debugEnable             = false;
masterConfig->ignoreAck               = false;
masterConfig->pinConfig               = kLPI2C_2PinOpenDrain;
masterConfig->baudRate_Hz             = 100000U;
masterConfig->busIdleTimeout_ns       = 0;
masterConfig->pinLowTimeout_ns        = 0;
masterConfig->sdaGlitchFilterWidth_ns = 0;
masterConfig->sclGlitchFilterWidth_ns = 0;
masterConfig->hostRequest.enable      = false;
masterConfig->hostRequest.source      = kLPI2C_HostRequestExternalPin;
masterConfig->hostRequest.polarity    = kLPI2C_HostRequestPinActiveHigh;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with LPI2C_MasterInit().

Parameters

- masterConfig – **[out]** User provided configuration structure for default values. Refer to lpi2c_master_config_t.

void LPI2C_MasterInit(LPI2C_Type *base, const *lpi2c_master_config_t* *masterConfig, uint32_t sourceClock_Hz)

Initializes the LPI2C master peripheral.

This function enables the peripheral clock and initializes the LPI2C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

- base – The LPI2C peripheral base address.
- masterConfig – User provided peripheral configuration. Use LPI2C_MasterGetDefaultConfig() to get a set of defaults that you can override.
- sourceClock_Hz – Frequency in Hertz of the LPI2C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

void LPI2C_MasterDeinit(LPI2C_Type *base)

Deinitializes the LPI2C master peripheral.

This function disables the LPI2C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- base – The LPI2C peripheral base address.

void LPI2C_MasterConfigureDataMatch(LPI2C_Type *base, const *lpi2c_data_match_config_t* *matchConfig)

Configures LPI2C master data match feature.

Parameters

- base – The LPI2C peripheral base address.
- matchConfig – Settings for the data match feature.

status_t LPI2C_MasterCheckAndClearError(LPI2C_Type *base, uint32_t status)

Convert provided flags to status code, and clear any errors if present.

Parameters

- base – The LPI2C peripheral base address.
- status – Current status flags value that will be checked.

Return values

- kStatus_Success –
- kStatus_LPI2C_PinLowTimeout –
- kStatus_LPI2C_ArbitrationLost –
- kStatus_LPI2C_Nak –
- kStatus_LPI2C_FifoError –

status_t LPI2C_CheckForBusyBus(LPI2C_Type *base)

Make sure the bus isn't already busy.

A busy bus is allowed if we are the one driving it.

Parameters

- base – The LPI2C peripheral base address.

Return values

- kStatus_Success –
- kStatus_LPI2C_Busy –

static inline void LPI2C_MasterReset(LPI2C_Type *base)

Performs a software reset.

Restores the LPI2C master peripheral to reset conditions.

Parameters

- base – The LPI2C peripheral base address.

static inline void LPI2C_MasterEnable(LPI2C_Type *base, bool enable)

Enables or disables the LPI2C module as master.

Parameters

- base – The LPI2C peripheral base address.
- enable – Pass true to enable or false to disable the specified LPI2C as master.

static inline uint32_t LPI2C_MasterGetStatusFlags(LPI2C_Type *base)

Gets the LPI2C master status flags.

A bit mask with the state of all LPI2C master status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

_lpi2c_master_flags

Parameters

- base – The LPI2C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

static inline void LPI2C_MasterClearStatusFlags(LPI2C_Type *base, uint32_t statusMask)

Clears the LPI2C master status flag state.

The following status register flags can be cleared:

- kLPI2C_MasterEndOfPacketFlag
- kLPI2C_MasterStopDetectFlag
- kLPI2C_MasterNackDetectFlag
- kLPI2C_MasterArbitrationLostFlag
- kLPI2C_MasterFifoErrFlag
- kLPI2C_MasterPinLowTimeoutFlag
- kLPI2C_MasterDataMatchFlag

Attempts to clear other flags has no effect.

See also:

_lpi2c_master_flags.

Parameters

- base – The LPI2C peripheral base address.
- statusMask – A bitmask of status flags that are to be cleared. The mask is composed of _lpi2c_master_flags enumerators OR'd together. You may pass the result of a previous call to LPI2C_MasterGetStatusFlags().

static inline void LPI2C_MasterEnableInterrupts(LPI2C_Type *base, uint32_t interruptMask)

Enables the LPI2C master interrupt requests.

All flags except kLPI2C_MasterBusyFlag and kLPI2C_MasterBusBusyFlag can be enabled as interrupts.

Parameters

- base – The LPI2C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See _lpi2c_master_flags for the set of constants that should be OR'd together to form the bit mask.

static inline void LPI2C_MasterDisableInterrupts(LPI2C_Type *base, uint32_t interruptMask)

Disables the LPI2C master interrupt requests.

All flags except kLPI2C_MasterBusyFlag and kLPI2C_MasterBusBusyFlag can be enabled as interrupts.

Parameters

- base – The LPI2C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See _lpi2c_master_flags for the set of constants that should be OR'd together to form the bit mask.

static inline uint32_t LPI2C_MasterGetEnabledInterrupts(LPI2C_Type *base)

Returns the set of currently enabled LPI2C master interrupt requests.

Parameters

- base – The LPI2C peripheral base address.

Returns

A bitmask composed of _lpi2c_master_flags enumerators OR'd together to indicate the set of enabled interrupts.

static inline void LPI2C_MasterEnableDMA(LPI2C_Type *base, bool enableTx, bool enableRx)

Enables or disables LPI2C master DMA requests.

Parameters

- base – The LPI2C peripheral base address.
- enableTx – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- enableRx – Enable flag for receive DMA request. Pass true for enable, false for disable.

static inline uint32_t LPI2C_MasterGetTxFifoAddress(LPI2C_Type *base)

Gets LPI2C master transmit data register address for DMA transfer.

Parameters

- base – The LPI2C peripheral base address.

Returns

The LPI2C Master Transmit Data Register address.

static inline uint32_t LPI2C_MasterGetRxFifoAddress(LPI2C_Type *base)

Gets LPI2C master receive data register address for DMA transfer.

Parameters

- base – The LPI2C peripheral base address.

Returns

The LPI2C Master Receive Data Register address.

static inline void LPI2C_MasterSetWatermarks(LPI2C_Type *base, size_t txWords, size_t rxWords)

Sets the watermarks for LPI2C master FIFOs.

Parameters

- base – The LPI2C peripheral base address.
- txWords – Transmit FIFO watermark value in words. The kLPI2C_MasterTxReadyFlag flag is set whenever the number of words in the transmit FIFO is equal or less than *txWords*. Writing a value equal or greater than the FIFO size is truncated.

- rxWords – Receive FIFO watermark value in words. The kLPI2C_MasterRxReadyFlag flag is set whenever the number of words in the receive FIFO is greater than *rxWords*. Writing a value equal or greater than the FIFO size is truncated.

static inline void LPI2C_MasterGetFifoCounts(LPI2C_Type *base, size_t *rxCount, size_t *txCount)

Gets the current number of words in the LPI2C master FIFOs.

Parameters

- base – The LPI2C peripheral base address.
- txCount – **[out]** Pointer through which the current number of words in the transmit FIFO is returned. Pass NULL if this value is not required.
- rxCount – **[out]** Pointer through which the current number of words in the receive FIFO is returned. Pass NULL if this value is not required.

void LPI2C_MasterSetBaudRate(LPI2C_Type *base, uint32_t sourceClock_Hz, uint32_t baudRate_Hz)

Sets the I2C bus frequency for master transactions.

The LPI2C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

Note: Please note that the second parameter is the clock frequency of LPI2C module, the third parameter means user configured bus baudrate, this implementation is different from other I2C drivers which use baudrate configuration as second parameter and source clock frequency as third parameter.

Parameters

- base – The LPI2C peripheral base address.
- sourceClock_Hz – LPI2C functional clock frequency in Hertz.
- baudRate_Hz – Requested bus frequency in Hertz.

static inline bool LPI2C_MasterGetBusIdleState(LPI2C_Type *base)

Returns whether the bus is idle.

Requires the master mode to be enabled.

Parameters

- base – The LPI2C peripheral base address.

Return values

- true – Bus is busy.
- false – Bus is idle.

status_t LPI2C_MasterStart(LPI2C_Type *base, uint8_t address, *lpi2c_direction_t* dir)

Sends a START signal and slave address on the I2C bus.

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the *address* parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

- base – The LPI2C peripheral base address.

- address – 7-bit slave device address, in bits [6:0].
- dir – Master transfer direction, either kLPI2C_Read or kLPI2C_Write. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

- kStatus_Success – START signal and address were successfully enqueued in the transmit FIFO.
- kStatus_LPI2C_Busy – Another master is currently utilizing the bus.

static inline *status_t* LPI2C_MasterRepeatedStart(LPI2C_Type *base, uint8_t address, *lpi2c_direction_t* dir)

Sends a repeated START signal and slave address on the I2C bus.

This function is used to send a Repeated START signal when a transfer is already in progress. Like LPI2C_MasterStart(), it also sends the specified 7-bit address.

Note: This function exists primarily to maintain compatible APIs between LPI2C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

- base – The LPI2C peripheral base address.
- address – 7-bit slave device address, in bits [6:0].
- dir – Master transfer direction, either kLPI2C_Read or kLPI2C_Write. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

- kStatus_Success – Repeated START signal and address were successfully enqueued in the transmit FIFO.
- kStatus_LPI2C_Busy – Another master is currently utilizing the bus.

status_t LPI2C_MasterSend(LPI2C_Type *base, void *txBuff, size_t txSize)

Performs a polling send transfer on the I2C bus.

Sends up to *txSize* number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns kStatus_LPI2C_Nak.

Parameters

- base – The LPI2C peripheral base address.
- txBuff – The pointer to the data to be transferred.
- txSize – The length in bytes of the data to be transferred.

Return values

- kStatus_Success – Data was sent successfully.
- kStatus_LPI2C_Busy – Another master is currently utilizing the bus.
- kStatus_LPI2C_Nak – The slave device sent a NAK in response to a byte.
- kStatus_LPI2C_FifoError – FIFO under run or over run.
- kStatus_LPI2C_ArbitrationLost – Arbitration lost error.
- kStatus_LPI2C_PinLowTimeout – SCL or SDA were held low longer than the timeout.

status_t LPI2C_MasterReceive(LPI2C_Type *base, void *rxBuff, size_t rxSize)

Performs a polling receive transfer on the I2C bus.

Parameters

- base – The LPI2C peripheral base address.
- rxBuff – The pointer to the data to be transferred.
- rxSize – The length in bytes of the data to be transferred.

Return values

- kStatus_Success – Data was received successfully.
- kStatus_LPI2C_Busy – Another master is currently utilizing the bus.
- kStatus_LPI2C_Nak – The slave device sent a NAK in response to a byte.
- kStatus_LPI2C_FifoError – FIFO under run or overrun.
- kStatus_LPI2C_ArbitrationLost – Arbitration lost error.
- kStatus_LPI2C_PinLowTimeout – SCL or SDA were held low longer than the timeout.

status_t LPI2C_MasterStop(LPI2C_Type *base)

Sends a STOP signal on the I2C bus.

This function does not return until the STOP signal is seen on the bus, or an error occurs.

Parameters

- base – The LPI2C peripheral base address.

Return values

- kStatus_Success – The STOP signal was successfully sent on the bus and the transaction terminated.
- kStatus_LPI2C_Busy – Another master is currently utilizing the bus.
- kStatus_LPI2C_Nak – The slave device sent a NAK in response to a byte.
- kStatus_LPI2C_FifoError – FIFO under run or overrun.
- kStatus_LPI2C_ArbitrationLost – Arbitration lost error.
- kStatus_LPI2C_PinLowTimeout – SCL or SDA were held low longer than the timeout.

status_t LPI2C_MasterTransferBlocking(LPI2C_Type *base, *lpi2c_master_transfer_t* *transfer)

Performs a master polling transfer on the I2C bus.

Note: The API does not return until the transfer succeeds or fails due to error happens during transfer.

Parameters

- base – The LPI2C peripheral base address.
- transfer – Pointer to the transfer structure.

Return values

- kStatus_Success – Data was received successfully.
- kStatus_LPI2C_Busy – Another master is currently utilizing the bus.
- kStatus_LPI2C_Nak – The slave device sent a NAK in response to a byte.

- kStatus_LPI2C_FifoError – FIFO under run or overrun.
- kStatus_LPI2C_ArbitrationLost – Arbitration lost error.
- kStatus_LPI2C_PinLowTimeout – SCL or SDA were held low longer than the timeout.

void LPI2C_MasterTransferCreateHandle(LPI2C_Type *base, *lpi2c_master_handle_t* *handle, *lpi2c_master_transfer_callback_t* callback, void *userData)

Creates a new handle for the LPI2C master non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the LPI2C_MasterTransferAbort() API shall be called.

Note: The function also enables the NVIC IRQ for the input LPI2C. Need to notice that on some SoCs the LPI2C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- base – The LPI2C peripheral base address.
- handle – **[out]** Pointer to the LPI2C master driver handle.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

status_t LPI2C_MasterTransferNonBlocking(LPI2C_Type *base, *lpi2c_master_handle_t* *handle, *lpi2c_master_transfer_t* *transfer)

Performs a non-blocking transaction on the I2C bus.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.
- transfer – The pointer to the transfer descriptor.

Return values

- kStatus_Success – The transaction was started successfully.
- kStatus_LPI2C_Busy – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

status_t LPI2C_MasterTransferGetCount(LPI2C_Type *base, *lpi2c_master_handle_t* *handle, size_t *count)

Returns number of bytes transferred so far.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.
- count – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

void LPI2C_MasterTransferAbort(LPI2C_Type *base, *lpi2c_master_handle_t* *handle)

Terminates a non-blocking LPI2C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the LPI2C peripheral's IRQ priority.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.

void LPI2C_MasterTransferHandleIRQ(LPI2C_Type *base, void *lpi2cMasterHandle)

Reusable routine to handle master interrupts.

Note: This function does not need to be called unless you are reimplementing the non-blocking API's interrupt handler routines to add special functionality.

Parameters

- base – The LPI2C peripheral base address.
- lpi2cMasterHandle – Pointer to the LPI2C master driver handle.

enum _lpi2c_master_flags

LPI2C master peripheral flags.

The following status register flags can be cleared:

- kLPI2C_MasterEndOfPacketFlag
- kLPI2C_MasterStopDetectFlag
- kLPI2C_MasterNackDetectFlag
- kLPI2C_MasterArbitrationLostFlag
- kLPI2C_MasterFifoErrFlag
- kLPI2C_MasterPinLowTimeoutFlag
- kLPI2C_MasterDataMatchFlag

All flags except kLPI2C_MasterBusyFlag and kLPI2C_MasterBusBusyFlag can be enabled as interrupts.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kLPI2C_MasterTxReadyFlag

Transmit data flag

enumerator kLPI2C_MasterRxReadyFlag

Receive data flag

enumerator kLPI2C_MasterEndOfPacketFlag

End Packet flag

enumerator kLPI2C_MasterStopDetectFlag

Stop detect flag

enumerator kLPI2C_MasterNackDetectFlag
NACK detect flag

enumerator kLPI2C_MasterArbitrationLostFlag
Arbitration lost flag

enumerator kLPI2C_MasterFifoErrFlag
FIFO error flag

enumerator kLPI2C_MasterPinLowTimeoutFlag
Pin low timeout flag

enumerator kLPI2C_MasterDataMatchFlag
Data match flag

enumerator kLPI2C_MasterBusyFlag
Master busy flag

enumerator kLPI2C_MasterBusBusyFlag
Bus busy flag

enumerator kLPI2C_MasterClearFlags
All flags which are cleared by the driver upon starting a transfer.

enumerator kLPI2C_MasterIrqFlags
IRQ sources enabled by the non-blocking transactional API.

enumerator kLPI2C_MasterErrorFlags
Errors to check for.

enum _lpi2c_direction
Direction of master and slave transfers.

Values:

enumerator kLPI2C_Write
Master transmit.

enumerator kLPI2C_Read
Master receive.

enum _lpi2c_master_pin_config
LPI2C pin configuration.

Values:

enumerator kLPI2C_2PinOpenDrain
LPI2C Configured for 2-pin open drain mode

enumerator kLPI2C_2PinOutputOnly
LPI2C Configured for 2-pin output only mode (ultra-fast mode)

enumerator kLPI2C_2PinPushPull
LPI2C Configured for 2-pin push-pull mode

enumerator kLPI2C_4PinPushPull
LPI2C Configured for 4-pin push-pull mode

enumerator kLPI2C_2PinOpenDrainWithSeparateSlave
LPI2C Configured for 2-pin open drain mode with separate LPI2C slave

enumerator kLPI2C_2PinOutputOnlyWithSeparateSlave
LPI2C Configured for 2-pin output only mode(ultra-fast mode) with separate LPI2C slave

enumerator kLPI2C_2PinPushPullWithSeparateSlave
LPI2C Configured for 2-pin push-pull mode with separate LPI2C slave

enumerator kLPI2C_4PinPushPullWithInvertedOutput
LPI2C Configured for 4-pin push-pull mode(inverted outputs)

enum _lpi2c_host_request_source
LPI2C master host request selection.

Values:

enumerator kLPI2C_HostRequestExternalPin
Select the LPI2C_HREQ pin as the host request input

enumerator kLPI2C_HostRequestInputTrigger
Select the input trigger as the host request input

enum _lpi2c_host_request_polarity
LPI2C master host request pin polarity configuration.

Values:

enumerator kLPI2C_HostRequestPinActiveLow
Configure the LPI2C_HREQ pin active low

enumerator kLPI2C_HostRequestPinActiveHigh
Configure the LPI2C_HREQ pin active high

enum _lpi2c_data_match_config_mode
LPI2C master data match configuration modes.

Values:

enumerator kLPI2C_MatchDisabled
LPI2C Match Disabled

enumerator kLPI2C_1stWordEqualsM0OrM1
LPI2C Match Enabled and 1st data word equals MATCH0 OR MATCH1

enumerator kLPI2C_AnyWordEqualsM0OrM1
LPI2C Match Enabled and any data word equals MATCH0 OR MATCH1

enumerator kLPI2C_1stWordEqualsM0And2ndWordEqualsM1
LPI2C Match Enabled and 1st data word equals MATCH0, 2nd data equals MATCH1

enumerator kLPI2C_AnyWordEqualsM0AndNextWordEqualsM1
LPI2C Match Enabled and any data word equals MATCH0, next data equals MATCH1

enumerator kLPI2C_1stWordAndM1EqualsM0AndM1
LPI2C Match Enabled and 1st data word and MATCH0 equals MATCH0 and MATCH1

enumerator kLPI2C_AnyWordAndM1EqualsM0AndM1
LPI2C Match Enabled and any data word and MATCH0 equals MATCH0 and MATCH1

enum _lpi2c_master_transfer_flags
Transfer option flags.

Note: These enumerations are intended to be OR'd together to form a bit mask of options for the _lpi2c_master_transfer::flags field.

Values:

enumerator kLPI2C_TransferDefaultFlag
Transfer starts with a start signal, stops with a stop signal.

enumerator kLPI2C_TransferNoStartFlag
Don't send a start condition, address, and sub address

enumerator kLPI2C_TransferRepeatedStartFlag
Send a repeated start condition

enumerator kLPI2C_TransferNoStopFlag
Don't send a stop condition.

typedef enum *_lpi2c_direction* lpi2c_direction_t
Direction of master and slave transfers.

typedef enum *_lpi2c_master_pin_config* lpi2c_master_pin_config_t
LPI2C pin configuration.

typedef enum *_lpi2c_host_request_source* lpi2c_host_request_source_t
LPI2C master host request selection.

typedef enum *_lpi2c_host_request_polarity* lpi2c_host_request_polarity_t
LPI2C master host request pin polarity configuration.

typedef struct *_lpi2c_master_config* lpi2c_master_config_t
Structure with settings to initialize the LPI2C master module.

This structure holds configuration settings for the LPI2C peripheral. To initialize this structure to reasonable defaults, call the LPI2C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

typedef enum *_lpi2c_data_match_config_mode* lpi2c_data_match_config_mode_t
LPI2C master data match configuration modes.

typedef struct *_lpi2c_match_config* lpi2c_data_match_config_t
LPI2C master data match configuration structure.

typedef struct *_lpi2c_master_transfer* lpi2c_master_transfer_t
LPI2C master descriptor of the transfer.

typedef struct *_lpi2c_master_handle* lpi2c_master_handle_t
LPI2C master handle of the transfer.

typedef void (*lpi2c_master_transfer_callback_t)(LPI2C_Type *base, *lpi2c_master_handle_t* *handle, *status_t* completionStatus, void *userData)

Master completion callback function pointer type.

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to LPI2C_MasterTransferCreateHandle().

Param base
The LPI2C peripheral base address.

Param handle
Pointer to the LPI2C master driver handle.

Param completionStatus
Either kStatus_Success or an error code describing how the transfer completed.

Param userData
Arbitrary pointer-sized value passed from the application.

typedef void (*lpi2c_master_isr_t)(LPI2C_Type *base, void *handle)

Typedef for master interrupt handler, used internally for LPI2C master interrupt and EDMA transactional APIs.

struct _lpi2c_master_config

#include <fsl_lpi2c.h> Structure with settings to initialize the LPI2C master module.

This structure holds configuration settings for the LPI2C peripheral. To initialize this structure to reasonable defaults, call the LPI2C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

bool enableMaster

Whether to enable master mode.

bool enableDoze

Whether master is enabled in doze mode.

bool debugEnable

Enable transfers to continue when halted in debug mode.

bool ignoreAck

Whether to ignore ACK/NACK.

lpi2c_master_pin_config_t pinConfig

The pin configuration option.

uint32_t baudRate_Hz

Desired baud rate in Hertz.

uint32_t busIdleTimeout_ns

Bus idle timeout in nanoseconds. Set to 0 to disable.

uint32_t pinLowTimeout_ns

Pin low timeout in nanoseconds. Set to 0 to disable.

uint8_t sdaGlitchFilterWidth_ns

Width in nanoseconds of glitch filter on SDA pin. Set to 0 to disable.

uint8_t sclGlitchFilterWidth_ns

Width in nanoseconds of glitch filter on SCL pin. Set to 0 to disable.

struct *_lpi2c_master_config* hostRequest

Host request options.

struct _lpi2c_match_config

#include <fsl_lpi2c.h> LPI2C master data match configuration structure.

Public Members

lpi2c_data_match_config_mode_t matchMode

Data match configuration setting.

bool rxDataMatchOnly

When set to true, received data is ignored until a successful match.

uint32_t match0

Match value 0.

uint32_t match1

Match value 1.

struct __lpi2c_master_transfer

#include <fsl_lpi2c.h> Non-blocking transfer descriptor structure.

This structure is used to pass transaction parameters to the LPI2C_MasterTransferNonBlocking() API.

Public Members

uint32_t flags

Bit mask of options for the transfer. See enumeration _lpi2c_master_transfer_flags for available options. Set to 0 or kLPI2C_TransferDefaultFlag for normal transfers.

uint16_t slaveAddress

The 7-bit slave address.

lpi2c_direction_t direction

Either kLPI2C_Read or kLPI2C_Write.

uint32_t subaddress

Sub address. Transferred MSB first.

size_t subaddressSize

Length of sub address to send in bytes. Maximum size is 4 bytes.

void *data

Pointer to data to transfer.

size_t dataSize

Number of bytes to transfer.

struct __lpi2c_master_handle

#include <fsl_lpi2c.h> Driver handle for master non-blocking APIs.

Note: The contents of this structure are private and subject to change.

Public Members

uint8_t state

Transfer state machine current state.

uint16_t remainingBytes

Remaining byte count in current state.

uint8_t *buf

Buffer pointer for current state.

uint16_t commandBuffer[6]

LPI2C command sequence. When all 6 command words are used: Start&addr&write[1 word] + subaddr[4 words] + restart&addr&read[1 word]

lpi2c_master_transfer_t transfer

Copy of the current transfer info.

lpi2c_master_transfer_callback_t completionCallback

Callback function pointer.

void *userData
Application data passed to callback.

struct hostRequest

Public Members

bool enable
Enable host request.

lpi2c_host_request_source_t source
Host request source.

lpi2c_host_request_polarity_t polarity
Host request pin polarity.

2.37 LPI2C Master DMA Driver

void LPI2C_MasterCreateEDMAHandle(LPI2C_Type *base, *lpi2c_master_edma_handle_t* *handle, *edma_handle_t* *rxDmaHandle, *edma_handle_t* *txDmaHandle, *lpi2c_master_edma_transfer_callback_t* callback, void *userData)

Create a new handle for the LPI2C master DMA APIs.

The creation of a handle is for use with the DMA APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the LPI2C_MasterTransferAbortEDMA() API shall be called.

For devices where the LPI2C send and receive DMA requests are OR'd together, the *txDmaHandle* parameter is ignored and may be set to NULL.

Parameters

- base – The LPI2C peripheral base address.
- handle – **[out]** Pointer to the LPI2C master driver handle.
- rxDmaHandle – Handle for the eDMA receive channel. Created by the user prior to calling this function.
- txDmaHandle – Handle for the eDMA transmit channel. Created by the user prior to calling this function.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

status_t LPI2C_MasterTransferEDMA(LPI2C_Type *base, *lpi2c_master_edma_handle_t* *handle, *lpi2c_master_transfer_t* *transfer)

Performs a non-blocking DMA-based transaction on the I2C bus.

The callback specified when the *handle* was created is invoked when the transaction has completed.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.
- transfer – The pointer to the transfer descriptor.

Return values

- kStatus_Success – The transaction was started successfully.
- kStatus_LPI2C_Busy – Either another master is currently utilizing the bus, or another DMA transaction is already in progress.

status_t LPI2C_MasterTransferGetCountEDMA(LPI2C_Type *base, *lpi2c_master_edma_handle_t* *handle, size_t *count)

Returns number of bytes transferred so far.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.
- count – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress – There is not a DMA transaction currently in progress.

status_t LPI2C_MasterTransferAbortEDMA(LPI2C_Type *base, *lpi2c_master_edma_handle_t* *handle)

Terminates a non-blocking LPI2C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the eDMA peripheral's IRQ priority.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to the LPI2C master driver handle.

Return values

- kStatus_Success – A transaction was successfully aborted.
- kStatus_LPI2C_Idle – There is not a DMA transaction currently in progress.

typedef struct *_lpi2c_master_edma_handle* lpi2c_master_edma_handle_t

LPI2C master EDMA handle of the transfer.

typedef void (*lpi2c_master_edma_transfer_callback_t)(LPI2C_Type *base, *lpi2c_master_edma_handle_t* *handle, *status_t* completionStatus, void *userData)

Master DMA completion callback function pointer type.

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to LPI2C_MasterCreateEDMAHandle().

Param base
The LPI2C peripheral base address.

Param handle
Handle associated with the completed transfer.

Param completionStatus
Either kStatus_Success or an error code describing how the transfer completed.

Param userData
Arbitrary pointer-sized value passed from the application.

struct __lpi2c_master_edma_handle
#include <fsl_lpi2c_edma.h> Driver handle for master DMA APIs.

Note: The contents of this structure are private and subject to change.

Public Members

LPI2C_Type *base
LPI2C base pointer.

bool isBusy
Transfer state machine current state.

uint8_t nbytes
eDMA minor byte transfer count initially configured.

uint16_t commandBuffer[20]
LPI2C command sequence. When all 10 command words are used: Start&addr&write[1 word] + subaddr[4 words] + restart&addr&read[1 word] + receive&Size[4 words]

lpi2c_master_transfer_t transfer
Copy of the current transfer info.

lpi2c_master_edma_transfer_callback_t completionCallback
Callback function pointer.

void *userData
Application data passed to callback.

edma_handle_t *rx
Handle for receive DMA channel.

edma_handle_t *tx
Handle for transmit DMA channel.

edma_tcd_t tcds[3]
Software TCD. Three are allocated to provide enough room to align to 32-bytes.

2.38 LPI2C Slave Driver

void LPI2C_SlaveGetDefaultConfig(*lpi2c_slave_config_t* *slaveConfig)
Provides a default configuration for the LPI2C slave peripheral.

This function provides the following default configuration for the LPI2C slave peripheral:

```
slaveConfig->enableSlave = true;
slaveConfig->address0 = 0U;
slaveConfig->address1 = 0U;
slaveConfig->addressMatchMode = kLPI2C_MatchAddress0;
slaveConfig->filterDozeEnable = true;
slaveConfig->filterEnable = true;
slaveConfig->enableGeneralCall = false;
slaveConfig->sclStall.enableAck = false;
```

(continues on next page)

(continued from previous page)

```
slaveConfig->sclStall.enableTx          = true;
slaveConfig->sclStall.enableRx          = true;
slaveConfig->sclStall.enableAddress     = true;
slaveConfig->ignoreAck                  = false;
slaveConfig->enableReceivedAddressRead = false;
slaveConfig->sdaGlitchFilterWidth_ns    = 0;
slaveConfig->sclGlitchFilterWidth_ns    = 0;
slaveConfig->dataValidDelay_ns          = 0;
slaveConfig->clockHoldTime_ns           = 0;
```

After calling this function, override any settings to customize the configuration, prior to initializing the master driver with LPI2C_SlaveInit(). Be sure to override at least the *address0* member of the configuration structure with the desired slave address.

Parameters

- slaveConfig – **[out]** User provided configuration structure that is set to default values. Refer to lpi2c_slave_config_t.

void LPI2C_SlaveInit(LPI2C_Type *base, const *lpi2c_slave_config_t* *slaveConfig, uint32_t sourceClock_Hz)

Initializes the LPI2C slave peripheral.

This function enables the peripheral clock and initializes the LPI2C slave peripheral as described by the user provided configuration.

Parameters

- base – The LPI2C peripheral base address.
- slaveConfig – User provided peripheral configuration. Use LPI2C_SlaveGetDefaultConfig() to get a set of defaults that you can override.
- sourceClock_Hz – Frequency in Hertz of the LPI2C functional clock. Used to calculate the filter widths, data valid delay, and clock hold time.

void LPI2C_SlaveDeinit(LPI2C_Type *base)

Deinitializes the LPI2C slave peripheral.

This function disables the LPI2C slave peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- base – The LPI2C peripheral base address.

static inline void LPI2C_SlaveReset(LPI2C_Type *base)

Performs a software reset of the LPI2C slave peripheral.

Parameters

- base – The LPI2C peripheral base address.

static inline void LPI2C_SlaveEnable(LPI2C_Type *base, bool enable)

Enables or disables the LPI2C module as slave.

Parameters

- base – The LPI2C peripheral base address.
- enable – Pass true to enable or false to disable the specified LPI2C as slave.

static inline uint32_t LPI2C_SlaveGetStatusFlags(LPI2C_Type *base)

Gets the LPI2C slave status flags.

A bit mask with the state of all LPI2C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

_lpi2c_slave_flags

Parameters

- base – The LPI2C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

static inline void LPI2C_SlaveClearStatusFlags(LPI2C_Type *base, uint32_t statusMask)

Clears the LPI2C status flag state.

The following status register flags can be cleared:

- kLPI2C_SlaveRepeatedStartDetectFlag
- kLPI2C_SlaveStopDetectFlag
- kLPI2C_SlaveBitErrFlag
- kLPI2C_SlaveFifoErrFlag

Attempts to clear other flags has no effect.

See also:

_lpi2c_slave_flags.

Parameters

- base – The LPI2C peripheral base address.
- statusMask – A bitmask of status flags that are to be cleared. The mask is composed of _lpi2c_slave_flags enumerators OR'd together. You may pass the result of a previous call to LPI2C_SlaveGetStatusFlags().

static inline void LPI2C_SlaveEnableInterrupts(LPI2C_Type *base, uint32_t interruptMask)

Enables the LPI2C slave interrupt requests.

All flags except kLPI2C_SlaveBusyFlag and kLPI2C_SlaveBusBusyFlag can be enabled as interrupts.

Parameters

- base – The LPI2C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See _lpi2c_slave_flags for the set of constants that should be OR'd together to form the bit mask.

static inline void LPI2C_SlaveDisableInterrupts(LPI2C_Type *base, uint32_t interruptMask)

Disables the LPI2C slave interrupt requests.

All flags except kLPI2C_SlaveBusyFlag and kLPI2C_SlaveBusBusyFlag can be enabled as interrupts.

Parameters

- base – The LPI2C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See _lpi2c_slave_flags for the set of constants that should be OR'd together to form the bit mask.

static inline uint32_t LPI2C_SlaveGetEnabledInterrupts(LPI2C_Type *base)

Returns the set of currently enabled LPI2C slave interrupt requests.

Parameters

- base – The LPI2C peripheral base address.

Returns

A bitmask composed of _lpi2c_slave_flags enumerators OR'd together to indicate the set of enabled interrupts.

static inline void LPI2C_SlaveEnableDMA(LPI2C_Type *base, bool enableAddressValid, bool enableRx, bool enableTx)

Enables or disables the LPI2C slave peripheral DMA requests.

Parameters

- base – The LPI2C peripheral base address.
- enableAddressValid – Enable flag for the address valid DMA request. Pass true for enable, false for disable. The address valid DMA request is shared with the receive data DMA request.
- enableRx – Enable flag for the receive data DMA request. Pass true for enable, false for disable.
- enableTx – Enable flag for the transmit data DMA request. Pass true for enable, false for disable.

static inline bool LPI2C_SlaveGetBusIdleState(LPI2C_Type *base)

Returns whether the bus is idle.

Requires the slave mode to be enabled.

Parameters

- base – The LPI2C peripheral base address.

Return values

- true – Bus is busy.
- false – Bus is idle.

static inline void LPI2C_SlaveTransmitAck(LPI2C_Type *base, bool ackOrNack)

Transmits either an ACK or NAK on the I2C bus in response to a byte from the master.

Use this function to send an ACK or NAK when the kLPI2C_SlaveTransmitAckFlag is asserted. This only happens if you enable the sclStall.enableAck field of the lpi2c_slave_config_t configuration structure used to initialize the slave peripheral.

Parameters

- base – The LPI2C peripheral base address.
- ackOrNack – Pass true for an ACK or false for a NAK.

static inline void LPI2C_SlaveEnableAckStall(LPI2C_Type *base, bool enable)

Enables or disables ACKSTALL.

When enables ACKSTALL, software can transmit either an ACK or NAK on the I2C bus in response to a byte from the master.

Parameters

- base – The LPI2C peripheral base address.
- enable – True will enable ACKSTALL,false will disable ACKSTALL.

static inline uint32_t LPI2C_SlaveGetReceivedAddress(LPI2C_Type *base)

Returns the slave address sent by the I2C master.

This function should only be called if the kLPI2C_SlaveAddressValidFlag is asserted.

Parameters

- base – The LPI2C peripheral base address.

Returns

The 8-bit address matched by the LPI2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

status_t LPI2C_SlaveSend(LPI2C_Type *base, void *txBuff, size_t txSize, size_t *actualTxSize)

Performs a polling send transfer on the I2C bus.

Parameters

- base – The LPI2C peripheral base address.
- txBuff – The pointer to the data to be transferred.
- txSize – The length in bytes of the data to be transferred.
- actualTxSize – **[out]**

Returns

Error or success status returned by API.

status_t LPI2C_SlaveReceive(LPI2C_Type *base, void *rxBuff, size_t rxSize, size_t *actualRxSize)

Performs a polling receive transfer on the I2C bus.

Parameters

- base – The LPI2C peripheral base address.
- rxBuff – The pointer to the data to be transferred.
- rxSize – The length in bytes of the data to be transferred.
- actualRxSize – **[out]**

Returns

Error or success status returned by API.

void LPI2C_SlaveTransferCreateHandle(LPI2C_Type *base, *lpi2c_slave_handle_t* *handle, *lpi2c_slave_transfer_callback_t* callback, void *userData)

Creates a new handle for the LPI2C slave non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the LPI2C_SlaveTransferAbort() API shall be called.

Note: The function also enables the NVIC IRQ for the input LPI2C. Need to notice that on some SoCs the LPI2C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- base – The LPI2C peripheral base address.
- handle – **[out]** Pointer to the LPI2C slave driver handle.

- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

status_t LPI2C_SlaveTransferNonBlocking(LPI2C_Type *base, *lpi2c_slave_handle_t* *handle, uint32_t eventMask)

Starts accepting slave transfers.

Call this API after calling I2C_SlaveInit() and LPI2C_SlaveTransferCreateHandle() to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to LPI2C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of lpi2c_slave_transfer_event_t enumerators for the events you wish to receive. The kLPI2C_SlaveTransmitEvent and kLPI2C_SlaveReceiveEvent events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the kLPI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to lpi2c_slave_handle_t structure which stores the transfer state.
- eventMask – Bit mask formed by OR'ing together lpi2c_slave_transfer_event_t enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and kLPI2C_SlaveAllEvents to enable all events.

Return values

- kStatus_Success – Slave transfers were successfully started.
- kStatus_LPI2C_Busy – Slave transfers have already been started on this handle.

status_t LPI2C_SlaveTransferGetCount(LPI2C_Type *base, *lpi2c_slave_handle_t* *handle, size_t *count)

Gets the slave transfer status during a non-blocking transfer.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to i2c_slave_handle_t structure.
- count – **[out]** Pointer to a value to hold the number of bytes transferred. May be NULL if the count is not required.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress –

void LPI2C_SlaveTransferAbort(LPI2C_Type *base, *lpi2c_slave_handle_t* *handle)

Aborts the slave non-blocking transfers.

Note: This API could be called at any time to stop slave for handling the bus events.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to lpi2c_slave_handle_t structure which stores the transfer state.

void LPI2C_SlaveTransferHandleIRQ(LPI2C_Type *base, *lpi2c_slave_handle_t* *handle)

Reusable routine to handle slave interrupts.

Note: This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

- base – The LPI2C peripheral base address.
- handle – Pointer to lpi2c_slave_handle_t structure which stores the transfer state.

enum _lpi2c_slave_flags

LPI2C slave peripheral flags.

The following status register flags can be cleared:

- kLPI2C_SlaveRepeatedStartDetectFlag
- kLPI2C_SlaveStopDetectFlag
- kLPI2C_SlaveBitErrFlag
- kLPI2C_SlaveFifoErrFlag

All flags except kLPI2C_SlaveBusyFlag and kLPI2C_SlaveBusBusyFlag can be enabled as interrupts.

Note: These enumerations are meant to be OR'd together to form a bit mask.

Values:

enumerator kLPI2C_SlaveTxReadyFlag

Transmit data flag

enumerator kLPI2C_SlaveRxReadyFlag

Receive data flag

enumerator kLPI2C_SlaveAddressValidFlag

Address valid flag

enumerator kLPI2C_SlaveTransmitAckFlag

Transmit ACK flag

enumerator kLPI2C_SlaveRepeatedStartDetectFlag

Repeated start detect flag

enumerator kLPI2C_SlaveStopDetectFlag

Stop detect flag

enumerator kLPI2C_SlaveBitErrFlag

Bit error flag

enumerator kLPI2C_SlaveFifoErrFlag

FIFO error flag

enumerator kLPI2C_SlaveAddressMatch0Flag
Address match 0 flag

enumerator kLPI2C_SlaveAddressMatch1Flag
Address match 1 flag

enumerator kLPI2C_SlaveGeneralCallFlag
General call flag

enumerator kLPI2C_SlaveBusyFlag
Master busy flag

enumerator kLPI2C_SlaveBusBusyFlag
Bus busy flag

enumerator kLPI2C_SlaveClearFlags
All flags which are cleared by the driver upon starting a transfer.

enumerator kLPI2C_SlaveIrqFlags
IRQ sources enabled by the non-blocking transactional API.

enumerator kLPI2C_SlaveErrorFlags
Errors to check for.

enum _lpi2c_slave_address_match
LPI2C slave address match options.

Values:

enumerator kLPI2C_MatchAddress0
Match only address 0.

enumerator kLPI2C_MatchAddress0OrAddress1
Match either address 0 or address 1.

enumerator kLPI2C_MatchAddress0ThroughAddress1
Match a range of slave addresses from address 0 through address 1.

enum _lpi2c_slave_transfer_event
Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to LPI2C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

Values:

enumerator kLPI2C_SlaveAddressMatchEvent
Received the slave address after a start or repeated start.

enumerator kLPI2C_SlaveTransmitEvent
Callback is requested to provide data to transmit (slave-transmitter role).

enumerator kLPI2C_SlaveReceiveEvent
Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator kLPI2C_SlaveTransmitAckEvent
Callback needs to either transmit an ACK or NACK.

enumerator kLPI2C_SlaveRepeatedStartEvent
A repeated start was detected.

enumerator kLPI2C_SlaveCompletionEvent
A stop was detected, completing the transfer.

enumerator kLPI2C_SlaveAllEvents
Bit mask of all available events.

typedef enum *_lpi2c_slave_address_match* lpi2c_slave_address_match_t
LPI2C slave address match options.

typedef struct *_lpi2c_slave_config* lpi2c_slave_config_t
Structure with settings to initialize the LPI2C slave module.

This structure holds configuration settings for the LPI2C slave peripheral. To initialize this structure to reasonable defaults, call the LPI2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

typedef enum *_lpi2c_slave_transfer_event* lpi2c_slave_transfer_event_t
Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to LPI2C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

typedef struct *_lpi2c_slave_transfer* lpi2c_slave_transfer_t
LPI2C slave transfer structure.

typedef struct *_lpi2c_slave_handle* lpi2c_slave_handle_t
LPI2C slave handle structure.

typedef void (*lpi2c_slave_transfer_callback_t)(LPI2C_Type *base, *lpi2c_slave_transfer_t* *transfer, void *userData)
Slave event callback function pointer type.

This callback is used only for the slave non-blocking transfer API. To install a callback, use the LPI2C_SlaveSetCallback() function after you have created a handle.

Param base
Base address for the LPI2C instance on which the event occurred.

Param transfer
Pointer to transfer descriptor containing values passed to and/or from the callback.

Param userData
Arbitrary pointer-sized value passed from the application.

struct _lpi2c_slave_config
#include <fsl_lpi2c.h> Structure with settings to initialize the LPI2C slave module.

This structure holds configuration settings for the LPI2C slave peripheral. To initialize this structure to reasonable defaults, call the LPI2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

bool enableSlave
: Enable slave mode.

uint8_t address0
: Slave's 7-bit address.

uint8_t address1
: Alternate slave 7-bit address.

lpi2c_slave_address_match_t addressMatchMode
: Address matching options.

bool filterDozeEnable
: Enable digital glitch filter in doze mode.

bool filterEnable
: Enable digital glitch filter.

bool enableGeneralCall
: Enable general call address matching.

struct *_lpi2c_slave_config* sclStall
: SCL stall enable options.

bool ignoreAck
: Continue transfers after a NACK is detected.

bool enableReceivedAddressRead
: Enable reading the address received address as the first byte of data.

uint32_t sdaGlitchFilterWidth_ns
: Width in nanoseconds of the digital filter on the SDA signal. Set to 0 to disable.

uint32_t sclGlitchFilterWidth_ns
: Width in nanoseconds of the digital filter on the SCL signal. Set to 0 to disable.

uint32_t dataValidDelay_ns
: Width in nanoseconds of the data valid delay.

uint32_t clockHoldTime_ns
: Width in nanoseconds of the clock hold time.

struct __lpi2c_slave_transfer
: *#include <fsl_lpi2c.h>* LPI2C slave transfer structure.

Public Members

lpi2c_slave_transfer_event_t event
: Reason the callback is being invoked.

uint8_t receivedAddress
: Matching address send by master.

uint8_t *data
: Transfer buffer

size_t dataSize
: Transfer size

status_t completionStatus

Success or error code describing how the transfer completed. Only applies for kLPI2C_SlaveCompletionEvent.

size_t transferredCount

Number of bytes actually transferred since start or last repeated start.

struct __lpi2c__slave__handle

#include <fsl_lpi2c.h> LPI2C slave handle structure.

Note: The contents of this structure are private and subject to change.

Public Members

lpi2c_slave_transfer_t transfer

LPI2C slave transfer copy.

bool isBusy

Whether transfer is busy.

bool wasTransmit

Whether the last transfer was a transmit.

uint32_t eventMask

Mask of enabled events.

uint32_t transferredCount

Count of bytes transferred.

lpi2c_slave_transfer_callback_t callback

Callback function called at transfer event.

void *userData

Callback parameter passed to callback.

struct sclStall

Public Members

bool enableAck

Enables SCL clock stretching during slave-transmit address byte(s) and slave-receiver address and data byte(s) to allow software to write the Transmit ACK Register before the ACK or NACK is transmitted. Clock stretching occurs when transmitting the 9th bit. When enableAckSCLStall is enabled, there is no need to set either enableRxDataSCLStall or enableAddressSCLStall.

bool enableTx

Enables SCL clock stretching when the transmit data flag is set during a slave-transmit transfer.

bool enableRx

Enables SCL clock stretching when receive data flag is set during a slave-receive transfer.

bool enableAddress

Enables SCL clock stretching when the address valid flag is asserted.

2.39 LPIT: Low-Power Interrupt Timer

void LPIT_Init(LPIT_Type *base, const *lpit_config_t* *config)

Ungates the LPIT clock and configures the peripheral for a basic operation.

This function issues a software reset to reset all channels and registers except the Module Control register.

Note: This API should be called at the beginning of the application using the LPIT driver.

Parameters

- base – LPIT peripheral base address.
- config – Pointer to the user configuration structure.

void LPIT_Deinit(LPIT_Type *base)

Disables the module and gates the LPIT clock.

Parameters

- base – LPIT peripheral base address.

void LPIT_GetDefaultConfig(*lpit_config_t* *config)

Fills in the LPIT configuration structure with default settings.

The default values are:

```
config->enableRunInDebug = false;
config->enableRunInDoze = false;
```

Parameters

- config – Pointer to the user configuration structure.

status_t LPIT_SetupChannel(LPIT_Type *base, *lpit_chnl_t* channel, const *lpit_chnl_params_t* *chnlSetup)

Sets up an LPIT channel based on the user's preference.

This function sets up the operation mode to one of the options available in the enumeration lpit_timer_modes_t. It sets the trigger source as either internal or external, trigger selection and the timers behaviour when a timeout occurs. It also chains the timer if a prior timer if requested by the user.

Parameters

- base – LPIT peripheral base address.
- channel – Channel that is being configured.
- chnlSetup – Configuration parameters.

static inline void LPIT_EnableInterrupts(LPIT_Type *base, uint32_t mask)

Enables the selected PIT interrupts.

Parameters

- base – LPIT peripheral base address.
- mask – The interrupts to enable. This is a logical OR of members of the enumeration lpit_interrupt_enable_t

static inline void LPIT_DisableInterrupts(LPIT_Type *base, uint32_t mask)

Disables the selected PIT interrupts.

Parameters

- base – LPIT peripheral base address.
- mask – The interrupts to enable. This is a logical OR of members of the enumeration lpit_interrupt_enable_t

static inline uint32_t LPIT_GetEnabledInterrupts(LPIT_Type *base)

Gets the enabled LPIT interrupts.

Parameters

- base – LPIT peripheral base address.

Returns

The enabled interrupts. This is the logical OR of members of the enumeration lpit_interrupt_enable_t

static inline uint32_t LPIT_GetStatusFlags(LPIT_Type *base)

Gets the LPIT status flags.

Parameters

- base – LPIT peripheral base address.

Returns

The status flags. This is the logical OR of members of the enumeration lpit_status_flags_t

static inline void LPIT_ClearStatusFlags(LPIT_Type *base, uint32_t mask)

Clears the LPIT status flags.

Parameters

- base – LPIT peripheral base address.
- mask – The status flags to clear. This is a logical OR of members of the enumeration lpit_status_flags_t

static inline void LPIT_SetTimerPeriod(LPIT_Type *base, *lpit_chnl_t* channel, uint32_t ticks)

Sets the timer period in units of count.

Timers begin counting down from the value set by this function until it reaches 0, at which point it generates an interrupt and loads this register value again. Writing a new value to this register does not restart the timer. Instead, the value is loaded after the timer expires.

Note: User can call the utility macros provided in fsl_common.h to convert to ticks.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.
- ticks – Timer period in units of ticks.

static inline void LPIT_SetTimerValue(LPIT_Type *base, *lpit_chnl_t* channel, uint32_t ticks)

Sets the timer period in units of count.

In the Dual 16-bit Periodic Counter mode, the counter will load and then the lower 16-bits will decrement down to zero, which will assert the output pre-trigger. The upper 16-bits will then decrement down to zero, which will negate the output pre-trigger and set the timer interrupt flag.

Note: Set TVAL register to 0 or 1 is invalid in compare mode.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.
- ticks – Timer period in units of ticks.

static inline uint32_t LPIT_GetCurrentTimerCount(LPIT_Type *base, *lpit_chnl_t* channel)

Reads the current timer counting value.

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note: User can call the utility macros provided in fsl_common.h to convert ticks to microseconds or milliseconds.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.

Returns

Current timer counting value in ticks.

static inline void LPIT_StartTimer(LPIT_Type *base, *lpit_chnl_t* channel)

Starts the timer counting.

After calling this function, timers load the period value and count down to 0. When the timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.

static inline void LPIT_StopTimer(LPIT_Type *base, *lpit_chnl_t* channel)

Stops the timer counting.

Parameters

- base – LPIT peripheral base address.
- channel – Timer channel number.

FSL_LPIT_DRIVER_VERSION

Version 2.1.3

enum _lpit_chnl

List of LPIT channels.

Note: Actual number of available channels is SoC-dependent

Values:

enumerator kLPIT_Chnl_0

LPIT channel number 0

enumerator kLPIT_Chnl_1
LPIT channel number 1

enumerator kLPIT_Chnl_2
LPIT channel number 2

enumerator kLPIT_Chnl_3
LPIT channel number 3

enum _lpit_timer_modes
Mode options available for the LPIT timer.

Values:

enumerator kLPIT_PeriodicCounter
Use the all 32-bits, counter loads and decrements to zero

enumerator kLPIT_DualPeriodicCounter
Counter loads, lower 16-bits decrement to zero, then upper 16-bits decrement

enumerator kLPIT_TriggerAccumulator
Counter loads on first trigger and decrements on each trigger

enumerator kLPIT_InputCapture
Counter loads with 0xFFFFFFFF, decrements to zero. It stores the inverse of the current value when a input trigger is detected

enum _lpit_trigger_select
Trigger options available.

This is used for both internal and external trigger sources. The actual trigger options available is SoC-specific, user should refer to the reference manual.

Values:

enumerator kLPIT_Trigger_TimerChn0
Channel 0 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn1
Channel 1 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn2
Channel 2 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn3
Channel 3 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn4
Channel 4 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn5
Channel 5 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn6
Channel 6 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn7
Channel 7 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn8
Channel 8 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn9
Channel 9 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn10
Channel 10 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn11
Channel 11 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn12
Channel 12 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn13
Channel 13 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn14
Channel 14 is selected as a trigger source

enumerator kLPIT_Trigger_TimerChn15
Channel 15 is selected as a trigger source

enum _lpit_trigger_source
Trigger source options available.

Values:

enumerator kLPIT_TriggerSource_External
Use external trigger input

enumerator kLPIT_TriggerSource_Internal
Use internal trigger

enum _lpit_interrupt_enable
List of LPIT interrupts.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

Values:

enumerator kLPIT_Channel0TimerInterruptEnable
Channel 0 Timer interrupt

enumerator kLPIT_Channel1TimerInterruptEnable
Channel 1 Timer interrupt

enumerator kLPIT_Channel2TimerInterruptEnable
Channel 2 Timer interrupt

enumerator kLPIT_Channel3TimerInterruptEnable
Channel 3 Timer interrupt

enum _lpit_status_flags
List of LPIT status flags.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

Values:

enumerator kLPIT_Channel0TimerFlag
Channel 0 Timer interrupt flag

enumerator kLPIT_Channel1TimerFlag
Channel 1 Timer interrupt flag

enumerator kLPIT_Channel2TimerFlag
Channel 2 Timer interrupt flag

enumerator kLPIT_Channel3TimerFlag
Channel 3 Timer interrupt flag

typedef enum *_lpit_chnl* lpit_chnl_t
List of LPIT channels.

Note: Actual number of available channels is SoC-dependent

typedef enum *_lpit_timer_modes* lpit_timer_modes_t
Mode options available for the LPIT timer.

typedef enum *_lpit_trigger_select* lpit_trigger_select_t
Trigger options available.

This is used for both internal and external trigger sources. The actual trigger options available is SoC-specific, user should refer to the reference manual.

typedef enum *_lpit_trigger_source* lpit_trigger_source_t
Trigger source options available.

typedef enum *_lpit_interrupt_enable* lpit_interrupt_enable_t
List of LPIT interrupts.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

typedef enum *_lpit_status_flags* lpit_status_flags_t
List of LPIT status flags.

Note: Number of timer channels are SoC-specific. See the SoC Reference Manual.

typedef struct *_lpit_chnl_params* lpit_chnl_params_t
Structure to configure the channel timer.

typedef struct *_lpit_config* lpit_config_t
LPIT configuration structure.

This structure holds the configuration settings for the LPIT peripheral. To initialize this structure to reasonable defaults, call the LPIT_GetDefaultConfig() function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

static void LPIT_ResetStateDelay(void)
Short wait for LPIT state reset.

After clear or set LPIT_EN, there should be delay longer than 4 LPIT functional clock.

static inline void LPIT_Reset(LPIT_Type *base)
Performs a software reset on the LPIT module.

This resets all channels and registers except the Module Control Register.

Parameters

- base – LPIT peripheral base address.

LPIT_RESET_STATE_DELAY

Delay used in LPIT_Reset.

The macro value should be larger than 4 * core clock / LPIT peripheral clock.

struct _lpit_chnl_params

#include <fsl_lpit.h> Structure to configure the channel timer.

Public Members

bool chainChannel

true: Timer chained to previous timer; false: Timer not chained

lpit_timer_modes_t timerMode

Timers mode of operation.

lpit_trigger_select_t triggerSelect

Trigger selection for the timer

lpit_trigger_source_t triggerSource

Decides if we use external or internal trigger.

bool enableReloadOnTrigger

true: Timer reloads when a trigger is detected; false: No effect

bool enableStopOnTimeout

true: Timer will stop after timeout; false: does not stop after timeout

bool enableStartOnTrigger

true: Timer starts when a trigger is detected; false: decrement immediately

struct _lpit_config

#include <fsl_lpit.h> LPIT configuration structure.

This structure holds the configuration settings for the LPIT peripheral. To initialize this structure to reasonable defaults, call the LPIT_GetDefaultConfig() function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

Public Members

bool enableRunInDebug

true: Timers run in debug mode; false: Timers stop in debug mode

bool enableRunInDoze

true: Timers run in doze mode; false: Timers stop in doze mode

2.40 LPSPI: Low Power Serial Peripheral Interface

2.41 LPSPI Peripheral driver

void LPSPI_MasterInit(LPSPI_Type *base, const *lpspi_master_config_t* *masterConfig, uint32_t srcClock_Hz)

Initializes the LPSPI master.

Parameters

- base – LPSPI peripheral address.
- masterConfig – Pointer to structure lpspi_master_config_t.
- srcClock_Hz – Module source input clock in Hertz

void LPSPI_MasterGetDefaultConfig(*lpspi_master_config_t* *masterConfig)

Sets the lpspi_master_config_t structure to default values.

This API initializes the configuration structure for LPSPI_MasterInit(). The initialized structure can remain unchanged in LPSPI_MasterInit(), or can be modified before calling the LPSPI_MasterInit(). Example:

```
lpspi_master_config_t masterConfig;
LPSPI_MasterGetDefaultConfig(&masterConfig);
```

Parameters

- masterConfig – pointer to lpspi_master_config_t structure

void LPSPI_SlaveInit(LPSPI_Type *base, const *lpspi_slave_config_t* *slaveConfig)

LPSPI slave configuration.

Parameters

- base – LPSPI peripheral address.
- slaveConfig – Pointer to a structure lpspi_slave_config_t.

void LPSPI_SlaveGetDefaultConfig(*lpspi_slave_config_t* *slaveConfig)

Sets the lpspi_slave_config_t structure to default values.

This API initializes the configuration structure for LPSPI_SlaveInit(). The initialized structure can remain unchanged in LPSPI_SlaveInit() or can be modified before calling the LPSPI_SlaveInit(). Example:

```
lpspi_slave_config_t slaveConfig;
LPSPI_SlaveGetDefaultConfig(&slaveConfig);
```

Parameters

- slaveConfig – pointer to lpspi_slave_config_t structure.

void LPSPI_Deinit(LPSPI_Type *base)

De-initializes the LPSPI peripheral. Call this API to disable the LPSPI clock.

Parameters

- base – LPSPI peripheral address.

void LPSPI_Reset(LPSPI_Type *base)

Restores the LPSPI peripheral to reset state. Note that this function sets all registers to reset state. As a result, the LPSPI module can't work after calling this API.

Parameters

- base – LPSPI peripheral address.

uint32_t LPSPI_GetInstance(LPSPI_Type *base)

Get the LPSPI instance from peripheral base address.

Parameters

- base – LPSPI peripheral base address.

Returns

LPSPI instance.

static inline void LPSPI_Enable(LPSPI_Type *base, bool enable)

Enables the LPSPI peripheral and sets the MCR MDIS to 0.

Parameters

- base – LPSPI peripheral address.
- enable – Pass true to enable module, false to disable module.

static inline uint32_t LPSPI_GetStatusFlags(LPSPI_Type *base)

Gets the LPSPI status flag state.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI status(in SR register).

static inline uint8_t LPSPI_GetTxFifoSize(LPSPI_Type *base)

Gets the LPSPI Tx FIFO size.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Tx FIFO size.

static inline uint8_t LPSPI_GetRxFifoSize(LPSPI_Type *base)

Gets the LPSPI Rx FIFO size.

Parameters

- base – LPSPI peripheral address.

Returns

The LPSPI Rx FIFO size.

static inline uint32_t LPSPI_GetTxFifoCount(LPSPI_Type *base)

Gets the LPSPI Tx FIFO count.

Parameters

- base – LPSPI peripheral address.

Returns

The number of words in the transmit FIFO.

static inline uint32_t LPSPI_GetRxFifoCount(LPSPI_Type *base)

Gets the LPSPI Rx FIFO count.

Parameters

- base – LPSPI peripheral address.

Returns

The number of words in the receive FIFO.

static inline void LPSPI_ClearStatusFlags(LPSPI_Type *base, uint32_t statusFlags)

Clears the LPSPI status flag.

This function clears the desired status bit by using a write-1-to-clear. The user passes in the base and the desired status flag bit to clear. The list of status flags is defined in the _lpspi_flags. Example usage:

```
LPSPI_ClearStatusFlags(base, kLPSPI_TxDataRequestFlag|kLPSPI_RxDataReadyFlag);
```

Parameters

- base – LPSPI peripheral address.
- statusFlags – The status flag used from type _lpspi_flags.

static inline uint32_t LPSPI_GetTcr(LPSPI_Type *base)

static inline void LPSPI_EnableInterrupts(LPSPI_Type *base, uint32_t mask)

Enables the LPSPI interrupts.

This function configures the various interrupt masks of the LPSPI. The parameters are base and an interrupt mask. Note that, for Tx fill and Rx FIFO drain requests, enabling the interrupt request disables the DMA request.

```
LPSPI_EnableInterrupts(base, kLPSPI_TxInterruptEnable | kLPSPI_RxInterruptEnable );
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum _lpspi_interrupt_enable.

static inline void LPSPI_DisableInterrupts(LPSPI_Type *base, uint32_t mask)

Disables the LPSPI interrupts.

```
LPSPI_DisableInterrupts(base, kLPSPI_TxInterruptEnable | kLPSPI_RxInterruptEnable );
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum _lpspi_interrupt_enable.

static inline void LPSPI_EnableDMA(LPSPI_Type *base, uint32_t mask)

Enables the LPSPI DMA request.

This function configures the Rx and Tx DMA mask of the LPSPI. The parameters are base and a DMA mask.

```
LPSPI_EnableDMA(base, kLPSPI_TxDmaEnable | kLPSPI_RxDmaEnable);
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum _lpspi_dma_enable.

static inline void LPSPI_DisableDMA(LPSPI_Type *base, uint32_t mask)

Disables the LPSPI DMA request.

This function configures the Rx and Tx DMA mask of the LPSPI. The parameters are base and a DMA mask.

```
SPI_DisableDMA(base, kLPSPI_TxDmaEnable | kLPSPI_RxDmaEnable);
```

Parameters

- base – LPSPI peripheral address.
- mask – The interrupt mask; Use the enum _lpspi_dma_enable.

static inline uint32_t LPSPI_GetTxRegisterAddress(LPSPI_Type *base)

Gets the LPSPI Transmit Data Register address for a DMA operation.

This function gets the LPSPI Transmit Data Register address because this value is needed for the DMA operation. This function can be used for either master or slave mode.

Parameters

- base – LPSPI peripheral address.

Returns
The LPSPI Transmit Data Register address.

static inline uint32_t LPSPI_GetRxRegisterAddress(LPSPI_Type *base)

Gets the LPSPI Receive Data Register address for a DMA operation.

This function gets the LPSPI Receive Data Register address because this value is needed for the DMA operation. This function can be used for either master or slave mode.

Parameters

- base – LPSPI peripheral address.

Returns
The LPSPI Receive Data Register address.

bool LPSPI_CheckTransferArgument(LPSPI_Type *base, *lpspi_transfer_t* *transfer, bool isEdma)

Check the argument for transfer .

Parameters

- base – LPSPI peripheral address.
- transfer – the transfer struct to be used.
- isEdma – True to check for EDMA transfer, false to check interrupt non-blocking transfer

Returns
Return true for right and false for wrong.

static inline void LPSPI_SetMasterSlaveMode(LPSPI_Type *base, *lpspi_master_slave_mode_t* mode)

Configures the LPSPI for either master or slave.

Note that the CFGR1 should only be written when the LPSPI is disabled (LPSPIx_CR_MEN = 0).

Parameters

- base – LPSPI peripheral address.
- mode – Mode setting (master or slave) of type lpspi_master_slave_mode_t.

static inline void LPSPI_SelectTransferPCS(LPSPI_Type *base, *lpspi_which_pcs_t* select)

Configures the peripheral chip select used for the transfer.

Parameters

- base – LPSPI peripheral address.
- select – LPSPI Peripheral Chip Select (PCS) configuration.

static inline void LPSPI_SetPCSContinous(LPSPI_Type *base, bool IsContinous)

Set the PCS signal to continuous or uncontinuous mode.

Note: In master mode, continuous transfer will keep the PCS asserted at the end of the frame size, until a command word is received that starts a new frame. So PCS must be set back to uncontinuous when transfer finishes. In slave mode, when continuous transfer is enabled, the LPSPI will only transmit the first frame size bits, after that the LPSPI will transmit received data back (assuming a 32-bit shift register).

Parameters

- base – LPSPI peripheral address.
- IsContinous – True to set the transfer PCS to continuous mode, false to set to uncontinuous mode.

static inline bool LPSPI_IsMaster(LPSPI_Type *base)

Returns whether the LPSPI module is in master mode.

Parameters

- base – LPSPI peripheral address.

Returns

Returns true if the module is in master mode or false if the module is in slave mode.

static inline void LPSPI_FlushFifo(LPSPI_Type *base, bool flushTxFifo, bool flushRxFifo)

Flushes the LPSPI FIFOs.

Parameters

- base – LPSPI peripheral address.
- flushTxFifo – Flushes (true) the Tx FIFO, else do not flush (false) the Tx FIFO.
- flushRxFifo – Flushes (true) the Rx FIFO, else do not flush (false) the Rx FIFO.

static inline void LPSPI_SetFifoWatermarks(LPSPI_Type *base, uint32_t txWater, uint32_t rxWater)

Sets the transmit and receive FIFO watermark values.

This function allows the user to set the receive and transmit FIFO watermarks. The function does not compare the watermark settings to the FIFO size. The FIFO watermark should not be equal to or greater than the FIFO size. It is up to the higher level driver to make this check.

Parameters

- base – LPSPI peripheral address.
- txWater – The TX FIFO watermark value. Writing a value equal or greater than the FIFO size is truncated.
- rxWater – The RX FIFO watermark value. Writing a value equal or greater than the FIFO size is truncated.

static inline void LPSPI_SetAllPcsPolarity(LPSPI_Type *base, uint32_t mask)

Configures all LPSPI peripheral chip select polarities simultaneously.

Note that the CFGR1 should only be written when the LPSPI is disabled (LPSPIx_CR_MEN = 0).

This is an example: PCS0 and PCS1 set to active low and other PCSs set to active high. Note that the number of PCS is device-specific.

```
LPSPI_SetAllPcsPolarity(base, kLPSPI_Pcs0ActiveLow | kLPSPI_Pcs1ActiveLow);
```

Parameters

- base – LPSPI peripheral address.
- mask – The PCS polarity mask; Use the enum _lpspi_pcs_polarity.

static inline void LPSPI_SetFrameSize(LPSPI_Type *base, uint32_t frameSize)

Configures the frame size.

The minimum frame size is 8-bits and the maximum frame size is 4096-bits. If the frame size is less than or equal to 32-bits, the word size and frame size are identical. If the frame

size is greater than 32-bits, the word size is 32-bits for each word except the last (the last word contains the remainder bits if the frame size is not divisible by 32). The minimum word size is 2-bits. A frame size of 33-bits (or similar) is not supported.

Note 1: The transmit command register should be initialized before enabling the LPSPI in slave mode, although the command register does not update until after the LPSPI is enabled. After it is enabled, the transmit command register should only be changed if the LPSPI is idle.

Note 2: The transmit and command FIFO is a combined FIFO that includes both transmit data and command words. That means the TCR register should be written to when the Tx FIFO is not full.

Parameters

- base – LPSPI peripheral address.
- frameSize – The frame size in number of bits.

uint32_t LPSPI_MasterSetBaudRate(LPSPI_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz, uint32_t *tcrPrescaleValue)

Sets the LPSPI baud rate in bits per second.

This function takes in the desired bitsPerSec (baud rate) and calculates the nearest possible baud rate without exceeding the desired baud rate and returns the calculated baud rate in bits-per-second. It requires the caller to provide the frequency of the module source clock (in Hertz). Note that the baud rate does not go into effect until the Transmit Control Register (TCR) is programmed with the prescale value. Hence, this function returns the prescale tcrPrescaleValue parameter for later programming in the TCR. The higher level peripheral driver should alert the user of an out of range baud rate input.

Note that the LPSPI module must first be disabled before configuring this. Note that the LPSPI module must be configured for master mode before configuring this.

Parameters

- base – LPSPI peripheral address.
- baudRate_Bps – The desired baud rate in bits per second.
- srcClock_Hz – Module source input clock in Hertz.
- tcrPrescaleValue – The TCR prescale value needed to program the TCR.

Returns

The actual calculated baud rate. This function may also return a "0" if the LPSPI is not configured for master mode or if the LPSPI module is not disabled.

void LPSPI_MasterSetDelayScaler(LPSPI_Type *base, uint32_t scaler, *lpspi_delay_type_t* whichDelay)

Manually configures a specific LPSPI delay parameter (module must be disabled to change the delay values).

This function configures the following: SCK to PCS delay, or PCS to SCK delay, or The configurations must occur between the transfer delay.

The delay names are available in type lpspi_delay_type_t.

The user passes the desired delay along with the delay value. This allows the user to directly set the delay values if they have pre-calculated them or if they simply wish to manually increment the value.

Note that the LPSPI module must first be disabled before configuring this. Note that the LPSPI module must be configured for master mode before configuring this.

Parameters

- base – LPSPI peripheral address.

- scaler – The 8-bit delay value 0x00 to 0xFF (255).
- whichDelay – The desired delay to configure, must be of type lpspi_delay_type_t.

uint32_t LPSPI_MasterSetDelayTimes(LPSPI_Type *base, uint32_t delayTimeInNanoSec, *lpspi_delay_type_t* whichDelay, uint32_t srcClock_Hz)

Calculates the delay based on the desired delay input in nanoseconds (module must be disabled to change the delay values).

This function calculates the values for the following: SCK to PCS delay, or PCS to SCK delay, or The configurations must occur between the transfer delay.

The delay names are available in type lpspi_delay_type_t.

The user passes the desired delay and the desired delay value in nano-seconds. The function calculates the value needed for the desired delay parameter and returns the actual calculated delay because an exact delay match may not be possible. In this case, the closest match is calculated without going below the desired delay value input. It is possible to input a very large delay value that exceeds the capability of the part, in which case the maximum supported delay is returned. It is up to the higher level peripheral driver to alert the user of an out of range delay input.

Note that the LPSPI module must be configured for master mode before configuring this. And note that the delayTime = LPSPI_clockSource / (PRESCALE * Delay_scaler).

Parameters

- base – LPSPI peripheral address.
- delayTimeInNanoSec – The desired delay value in nano-seconds.
- whichDelay – The desired delay to configuration, which must be of type lpspi_delay_type_t.
- srcClock_Hz – Module source input clock in Hertz.

Returns

actual Calculated delay value in nano-seconds.

static inline void LPSPI_WriteData(LPSPI_Type *base, uint32_t data)

Writes data into the transmit data buffer.

This function writes data passed in by the user to the Transmit Data Register (TDR). The user can pass up to 32-bits of data to load into the TDR. If the frame size exceeds 32-bits, the user has to manage sending the data one 32-bit word at a time. Any writes to the TDR result in an immediate push to the transmit FIFO. This function can be used for either master or slave modes.

Parameters

- base – LPSPI peripheral address.
- data – The data word to be sent.

static inline uint32_t LPSPI_ReadData(LPSPI_Type *base)

Reads data from the data buffer.

This function reads the data from the Receive Data Register (RDR). This function can be used for either master or slave mode.

Parameters

- base – LPSPI peripheral address.

Returns

The data read from the data buffer.

void LPSPI_SetDummyData(LPSPI_Type *base, uint8_t dummyData)

Set up the dummy data.

Parameters

- base – LPSPI peripheral address.
- dummyData – Data to be transferred when tx buffer is NULL. Note: This API has no effect when LPSPI in slave interrupt mode, because driver will set the TXMSK bit to 1 if txData is NULL, no data is loaded from transmit FIFO and output pin is tristated.

void LPSPI_MasterTransferCreateHandle(LPSPI_Type *base, *lpspi_master_handle_t* *handle, *lpspi_master_transfer_callback_t* callback, void *userData)

Initializes the LPSPI master handle.

This function initializes the LPSPI handle, which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Parameters

- base – LPSPI peripheral address.
- handle – LPSPI handle pointer to lpspi_master_handle_t.
- callback – DSPI callback.
- userData – callback function parameter.

status_t LPSPI_MasterTransferBlocking(LPSPI_Type *base, *lpspi_transfer_t* *transfer)

LPSPI master transfer data using a polling method.

This function transfers data using a polling method. This is a blocking function, which does not return until all transfers have been completed.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not integer multiples of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral address.
- transfer – pointer to lpspi_transfer_t structure.

Returns

status of status_t.

status_t LPSPI_MasterTransferNonBlocking(LPSPI_Type *base, *lpspi_master_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI master transfer data using an interrupt method.

This function transfers data using an interrupt method. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not integer multiples of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to lpspi_master_handle_t structure which stores the transfer state.

- transfer – pointer to lpspi_transfer_t structure.

Returns
status of status_t.

status_t LPSPI_MasterTransferGetCount(LPSPI_Type *base, *lpspi_master_handle_t* *handle, size_t *count)

Gets the master transfer remaining bytes.

This function gets the master transfer remaining bytes.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to lpspi_master_handle_t structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Returns
status of status_t.

void LPSPI_MasterTransferAbort(LPSPI_Type *base, *lpspi_master_handle_t* *handle)

LPSPI master abort transfer which uses an interrupt method.

This function aborts a transfer which uses an interrupt method.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to lpspi_master_handle_t structure which stores the transfer state.

void LPSPI_MasterTransferHandleIRQ(LPSPI_Type *base, *lpspi_master_handle_t* *handle)

LPSPI Master IRQ handler function.

This function processes the LPSPI transmit and receive IRQ.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to lpspi_master_handle_t structure which stores the transfer state.

void LPSPI_SlaveTransferCreateHandle(LPSPI_Type *base, *lpspi_slave_handle_t* *handle, *lpspi_slave_transfer_callback_t* callback, void *userData)

Initializes the LPSPI slave handle.

This function initializes the LPSPI handle, which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Parameters

- base – LPSPI peripheral address.
- handle – LPSPI handle pointer to lpspi_slave_handle_t.
- callback – DSPI callback.
- userData – callback function parameter.

status_t LPSPI_SlaveTransferNonBlocking(LPSPI_Type *base, *lpspi_slave_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI slave transfer data using an interrupt method.

This function transfer data using an interrupt method. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to lpspi_slave_handle_t structure which stores the transfer state.
- transfer – pointer to lpspi_transfer_t structure.

Returns
status of status_t.

status_t LPSPI_SlaveTransferGetCount(LPSPI_Type *base, *lpspi_slave_handle_t* *handle, size_t *count)

Gets the slave transfer remaining bytes.

This function gets the slave transfer remaining bytes.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to lpspi_slave_handle_t structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Returns
status of status_t.

void LPSPI_SlaveTransferAbort(LPSPI_Type *base, *lpspi_slave_handle_t* *handle)

LPSPI slave aborts a transfer which uses an interrupt method.

This function aborts a transfer which uses an interrupt method.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to lpspi_slave_handle_t structure which stores the transfer state.

void LPSPI_SlaveTransferHandleIRQ(LPSPI_Type *base, *lpspi_slave_handle_t* *handle)

LPSPI Slave IRQ handler function.

This function processes the LPSPI transmit and receives an IRQ.

Parameters

- base – LPSPI peripheral address.
- handle – pointer to lpspi_slave_handle_t structure which stores the transfer state.

bool LPSPI_WaitTxFifoEmpty(LPSPI_Type *base)

Wait for tx FIFO to be empty.

This function wait the tx fifo empty

Parameters

- base – LPSPI peripheral address.

Returns
true for the tx FIFO is ready, false is not.

void LPSPI_DriverIRQHandler(uint32_t instance)

LPSPI driver IRQ handler common entry.

This function provides the common IRQ request entry for LPSPI.

Parameters

- instance – LPSPI instance.

FSL_LPSPI_DRIVER_VERSION

LPSPI driver version.

Status for the LPSPI driver.

Values:

enumerator kStatus_LPSPI_Busy

LPSPI transfer is busy.

enumerator kStatus_LPSPI_Error

LPSPI driver error.

enumerator kStatus_LPSPI_Idle

LPSPI is idle.

enumerator kStatus_LPSPI_OutOfRange

LPSPI transfer out Of range.

enumerator kStatus_LPSPI_Timeout

LPSPI timeout polling status flags.

enum _lpspi_flags

LPSPI status flags in SPIx_SR register.

Values:

enumerator kLPSPI_TxDataRequestFlag

Transmit data flag

enumerator kLPSPI_RxDataReadyFlag

Receive data flag

enumerator kLPSPI_WordCompleteFlag

Word Complete flag

enumerator kLPSPI_FrameCompleteFlag

Frame Complete flag

enumerator kLPSPI_TransferCompleteFlag

Transfer Complete flag

enumerator kLPSPI_TransmitErrorFlag

Transmit Error flag (FIFO underrun)

enumerator kLPSPI_ReceiveErrorFlag

Receive Error flag (FIFO overrun)

enumerator kLPSPI_DataMatchFlag

Data Match flag

enumerator kLPSPI_ModuleBusyFlag

Module Busy flag

enumerator kLPSPI_AllStatusFlag
Used for clearing all w1c status flags

enum _lpspi_interrupt_enable
LPSPI interrupt source.

Values:

enumerator kLPSPI_TxInterruptEnable
Transmit data interrupt enable

enumerator kLPSPI_RxInterruptEnable
Receive data interrupt enable

enumerator kLPSPI_WordCompleteInterruptEnable
Word complete interrupt enable

enumerator kLPSPI_FrameCompleteInterruptEnable
Frame complete interrupt enable

enumerator kLPSPI_TransferCompleteInterruptEnable
Transfer complete interrupt enable

enumerator kLPSPI_TransmitErrorInterruptEnable
Transmit error interrupt enable(FIFO underrun)

enumerator kLPSPI_ReceiveErrorInterruptEnable
Receive Error interrupt enable (FIFO overrun)

enumerator kLPSPI_DataMatchInterruptEnable
Data Match interrupt enable

enumerator kLPSPI_AllInterruptEnable
All above interrupts enable.

enum _lpspi_dma_enable
LPSPI DMA source.

Values:

enumerator kLPSPI_TxDmaEnable
Transmit data DMA enable

enumerator kLPSPI_RxDmaEnable
Receive data DMA enable

enum _lpspi_master_slave_mode
LPSPI master or slave mode configuration.

Values:

enumerator kLPSPI_Master
LPSPI peripheral operates in master mode.

enumerator kLPSPI_Slave
LPSPI peripheral operates in slave mode.

enum _lpspi_which_pcs_config
LPSPI Peripheral Chip Select (PCS) configuration (which PCS to configure).

Values:

enumerator kLPSPI_Pcs0
PCS[0]

enumerator kLPSPI_Pcs1
PCS[1]

enumerator kLPSPI_Pcs2
PCS[2]

enumerator kLPSPI_Pcs3
PCS[3]

enum _lpspi_pcs_polarity_config
LPSPI Peripheral Chip Select (PCS) Polarity configuration.

Values:

enumerator kLPSPI_PcsActiveHigh
PCS Active High (idles low)

enumerator kLPSPI_PcsActiveLow
PCS Active Low (idles high)

enum _lpspi_pcs_polarity
LPSPI Peripheral Chip Select (PCS) Polarity.

Values:

enumerator kLPSPI_Pcs0ActiveLow
Pcs0 Active Low (idles high).

enumerator kLPSPI_Pcs1ActiveLow
Pcs1 Active Low (idles high).

enumerator kLPSPI_Pcs2ActiveLow
Pcs2 Active Low (idles high).

enumerator kLPSPI_Pcs3ActiveLow
Pcs3 Active Low (idles high).

enumerator kLPSPI_PcsAllActiveLow
Pcs0 to Pcs5 Active Low (idles high).

enum _lpspi_clock_polarity
LPSPI clock polarity configuration.

Values:

enumerator kLPSPI_ClockPolarityActiveHigh
CPOL=0. Active-high LPSPI clock (idles low)

enumerator kLPSPI_ClockPolarityActiveLow
CPOL=1. Active-low LPSPI clock (idles high)

enum _lpspi_clock_phase
LPSPI clock phase configuration.

Values:

enumerator kLPSPI_ClockPhaseFirstEdge
CPHA=0. Data is captured on the leading edge of the SCK and changed on the following edge.

enumerator kLPSPI_ClockPhaseSecondEdge
CPHA=1. Data is changed on the leading edge of the SCK and captured on the following edge.

enum _lpspi_shift_direction

LPSPI data shifter direction options.

Values:

enumerator kLPSPI_MsbFirst

Data transfers start with most significant bit.

enumerator kLPSPI_LsbFirst

Data transfers start with least significant bit.

enum _lpspi_host_request_select

LPSPI Host Request select configuration.

Values:

enumerator kLPSPI_HostReqExtPin

Host Request is an ext pin.

enumerator kLPSPI_HostReqInternalTrigger

Host Request is an internal trigger.

enum _lpspi_match_config

LPSPI Match configuration options.

Values:

enumerator kLPSI_MatchDisabled

LPSPI Match Disabled.

enumerator kLPSI_1stWordEqualsM0orM1

LPSPI Match Enabled.

enumerator kLPSI_AnyWordEqualsM0orM1

LPSPI Match Enabled.

enumerator kLPSI_1stWordEqualsM0and2ndWordEqualsM1

LPSPI Match Enabled.

enumerator kLPSI_AnyWordEqualsM0andNxtWordEqualsM1

LPSPI Match Enabled.

enumerator kLPSI_1stWordAndM1EqualsM0andM1

LPSPI Match Enabled.

enumerator kLPSI_AnyWordAndM1EqualsM0andM1

LPSPI Match Enabled.

enum _lpspi_pin_config

LPSPI pin (SDO and SDI) configuration.

Values:

enumerator kLPSPI_SdiInSdoOut

LPSPI SDI input, SDO output.

enumerator kLPSPI_SdiInSdiOut

LPSPI SDI input, SDI output.

enumerator kLPSPI_SdoInSdoOut

LPSPI SDO input, SDO output.

enumerator kLPSPI_SdoInSdiOut

LPSPI SDO input, SDI output.

enum _lpspi_data_out_config

LPSPI data output configuration.

Values:

enumerator kLpspiDataOutRetained

Data out retains last value when chip select is de-asserted

enumerator kLpspiDataOutTristate

Data out is tristated when chip select is de-asserted

enum _lpspi_transfer_width

LPSPI transfer width configuration.

Values:

enumerator kLPSPI_SingleBitXfer

1-bit shift at a time, data out on SDO, in on SDI (normal mode)

enumerator kLPSPI_TwoBitXfer

2-bits shift out on SDO/SDI and in on SDO/SDI

enumerator kLPSPI_FourBitXfer

4-bits shift out on SDO/SDI/PCS[3:2] and in on SDO/SDI/PCS[3:2]

enum _lpspi_delay_type

LPSPI delay type selection.

Values:

enumerator kLPSPI_PcsToSck

PCS-to-SCK delay.

enumerator kLPSPI_LastSckToPcs

Last SCK edge to PCS delay.

enumerator kLPSPI_BetweenTransfer

Delay between transfers.

enum _lpspi_transfer_config_flag_for_master

Use this enumeration for LPSPI master transfer configFlags.

Values:

enumerator kLPSPI_MasterPcs0

LPSPI master PCS shift macro , internal used. LPSPI master transfer use PCS0 signal

enumerator kLPSPI_MasterPcs1

LPSPI master PCS shift macro , internal used. LPSPI master transfer use PCS1 signal

enumerator kLPSPI_MasterPcs2

LPSPI master PCS shift macro , internal used. LPSPI master transfer use PCS2 signal

enumerator kLPSPI_MasterPcs3

LPSPI master PCS shift macro , internal used. LPSPI master transfer use PCS3 signal

enumerator kLPSPI_MasterPcsContinuous

Is PCS signal continuous

enumerator kLPSPI_MasterByteSwap

Is master swap the byte. For example, when want to send data 1 2 3 4 5 6 7 8 (suppose you set lpspi_shift_direction_t to MSB).

i. If you set bitPerFrame = 8 , no matter the kLPSPI_MasterByteSwapyou flag is used or not, the waveform is 1 2 3 4 5 6 7 8.

ii. If you set bitPerFrame = 16 : (1) the waveform is 2 1 4 3 6 5 8 7 if you do not use the kLPSPI_MasterByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPI_MasterByteSwap flag.

iii. If you set bitPerFrame = 32 : (1) the waveform is 4 3 2 1 8 7 6 5 if you do not use the kLPSPI_MasterByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPI_MasterByteSwap flag.

enum _lpspi_transfer_config_flag_for_slave

Use this enumeration for LPSPI slave transfer configFlags.

Values:

enumerator kLPSPI_SlavePcs0

LPSPI slave PCS shift macro , internal used. LPSPI slave transfer use PCS0 signal

enumerator kLPSPI_SlavePcs1

LPSPI slave PCS shift macro , internal used. LPSPI slave transfer use PCS1 signal

enumerator kLPSPI_SlavePcs2

LPSPI slave PCS shift macro , internal used. LPSPI slave transfer use PCS2 signal

enumerator kLPSPI_SlavePcs3

LPSPI slave PCS shift macro , internal used. LPSPI slave transfer use PCS3 signal

enumerator kLPSPI_SlaveByteSwap

Is slave swap the byte. For example, when want to send data 1 2 3 4 5 6 7 8 (suppose you set lpspi_shift_direction_t to MSB).

i. If you set bitPerFrame = 8 , no matter the kLPSPI_SlaveByteSwap flag is used or not, the waveform is 1 2 3 4 5 6 7 8.

ii. If you set bitPerFrame = 16 : (1) the waveform is 2 1 4 3 6 5 8 7 if you do not use the kLPSPI_SlaveByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPI_SlaveByteSwap flag.

iii. If you set bitPerFrame = 32 : (1) the waveform is 4 3 2 1 8 7 6 5 if you do not use the kLPSPI_SlaveByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPI_SlaveByteSwap flag.

enum _lpspi_transfer_state

LPSPI transfer state, which is used for LPSPI transactional API state machine.

Values:

enumerator kLPSPI_Idle

Nothing in the transmitter/receiver.

enumerator kLPSPI_Busy

Transfer queue is not finished.

enumerator kLPSPI_Error

Transfer error.

typedef enum *_lpspi_master_slave_mode* lpspi_master_slave_mode_t

LPSPI master or slave mode configuration.

typedef enum *_lpspi_which_pcs_config* lpspi_which_pcs_t

LPSPI Peripheral Chip Select (PCS) configuration (which PCS to configure).

typedef enum *_lpspi_pcs_polarity_config* lpspi_pcs_polarity_config_t

LPSPI Peripheral Chip Select (PCS) Polarity configuration.

typedef enum *_lpspi_clock_polarity* lpspi_clock_polarity_t
LPSPI clock polarity configuration.

typedef enum *_lpspi_clock_phase* lpspi_clock_phase_t
LPSPI clock phase configuration.

typedef enum *_lpspi_shift_direction* lpspi_shift_direction_t
LPSPI data shifter direction options.

typedef enum *_lpspi_host_request_select* lpspi_host_request_select_t
LPSPI Host Request select configuration.

typedef enum *_lpspi_match_config* lpspi_match_config_t
LPSPI Match configuration options.

typedef enum *_lpspi_pin_config* lpspi_pin_config_t
LPSPI pin (SDO and SDI) configuration.

typedef enum *_lpspi_data_out_config* lpspi_data_out_config_t
LPSPI data output configuration.

typedef enum *_lpspi_transfer_width* lpspi_transfer_width_t
LPSPI transfer width configuration.

typedef enum *_lpspi_delay_type* lpspi_delay_type_t
LPSPI delay type selection.

typedef struct *_lpspi_master_config* lpspi_master_config_t
LPSPI master configuration structure.

typedef struct *_lpspi_slave_config* lpspi_slave_config_t
LPSPI slave configuration structure.

typedef struct *_lpspi_master_handle* lpspi_master_handle_t
Forward declaration of the _lpspi_master_handle typedefs.

typedef struct *_lpspi_slave_handle* lpspi_slave_handle_t
Forward declaration of the _lpspi_slave_handle typedefs.

typedef void (*lpspi_master_transfer_callback_t)(LPSPI_Type *base, *lpspi_master_handle_t* *handle, *status_t* status, void *userData)
Master completion callback function pointer type.

Param base
LPSPI peripheral address.

Param handle
Pointer to the handle for the LPSPI master.

Param status
Success or error code describing whether the transfer is completed.

Param userData
Arbitrary pointer-dataSized value passed from the application.

typedef void (*lpspi_slave_transfer_callback_t)(LPSPI_Type *base, *lpspi_slave_handle_t* *handle, *status_t* status, void *userData)
Slave completion callback function pointer type.

Param base
LPSPI peripheral address.

Param handle
Pointer to the handle for the LPSPI slave.

Param status
Success or error code describing whether the transfer is completed.

Param userData
Arbitrary pointer-dataSized value passed from the application.

typedef struct *_lpspi_transfer* lpspi_transfer_t
LPSPI master/slave transfer structure.

volatile uint8_t g_lpspiDummyData[]
Global variable for dummy data value setting.

LPSPI_DUMMY_DATA
LPSPI dummy data if no Tx data.

Dummy data used for tx if there is not txData.

SPI_RETRY_TIMES
Retry times for waiting flag.

LPSPI_MASTER_PCS_SHIFT
LPSPI master PCS shift macro , internal used.

LPSPI_MASTER_PCS_MASK
LPSPI master PCS shift macro , internal used.

LPSPI_SLAVE_PCS_SHIFT
LPSPI slave PCS shift macro , internal used.

LPSPI_SLAVE_PCS_MASK
LPSPI slave PCS shift macro , internal used.

struct _lpspi_master_config
#include <fsl_lpspi.h> LPSPI master configuration structure.

Public Members

uint32_t baudRate
Baud Rate for LPSPI.

uint32_t bitsPerFrame
Bits per frame, minimum 8, maximum 4096.

lpspi_clock_polarity_t cpol
Clock polarity.

lpspi_clock_phase_t cpha
Clock phase.

lpspi_shift_direction_t direction
MSB or LSB data shift direction.

uint32_t pcsToSckDelayInNanoSec
PCS to SCK delay time in nanoseconds, setting to 0 sets the minimum delay. It sets the boundary value if out of range.

uint32_t lastSckToPcsDelayInNanoSec
Last SCK to PCS delay time in nanoseconds, setting to 0 sets the minimum delay. It sets the boundary value if out of range.

uint32_t betweenTransferDelayInNanoSec
After the SCK delay time with nanoseconds, setting to 0 sets the minimum delay. It sets the boundary value if out of range.

lpspi_which_pcs_t whichPcs
Desired Peripheral Chip Select (PCS).

lpspi_pcs_polarity_config_t pcsActiveHighOrLow
Desired PCS active high or low

lpspi_pin_config_t pinCfg
Configures which pins are used for input and output data during single bit transfers.

lpspi_data_out_config_t dataOutConfig
Configures if the output data is tristated between accesses (LPSPI_PCS is negated).

bool enableInputDelay
Enable master to sample the input data on a delayed SCK. This can help improve slave setup time. Refer to device data sheet for specific time length.

struct __lpspi__slave__config
#include <fsl_lpspi.h> LPSPI slave configuration structure.

Public Members

uint32_t bitsPerFrame
Bits per frame, minimum 8, maximum 4096.

lpspi_clock_polarity_t cpol
Clock polarity.

lpspi_clock_phase_t cpha
Clock phase.

lpspi_shift_direction_t direction
MSB or LSB data shift direction.

lpspi_which_pcs_t whichPcs
Desired Peripheral Chip Select (pcs)

lpspi_pcs_polarity_config_t pcsActiveHighOrLow
Desired PCS active high or low

lpspi_pin_config_t pinCfg
Configures which pins are used for input and output data during single bit transfers.

lpspi_data_out_config_t dataOutConfig
Configures if the output data is tristated between accesses (LPSPI_PCS is negated).

struct __lpspi__transfer
#include <fsl_lpspi.h> LPSPI master/slave transfer structure.

Public Members

const uint8_t *txData
Send buffer.

uint8_t *rxData
Receive buffer.

volatile size_t dataSize
Transfer bytes.

uint32_t configFlags
Transfer transfer configuration flags. Set from _lpspi_transfer_config_flag_for_master if the transfer is used for master or _lpspi_transfer_config_flag_for_slave enumeration if the transfer is used for slave.

struct __lpspi__master__handle
#include <fsl_lpspi.h> LPSPI master transfer handle structure used for transactional API.

Public Members

volatile bool isPcsContinuous
Is PCS continuous in transfer.

volatile bool writeTcrInIsr
A flag that whether should write TCR in ISR.

volatile bool isByteSwap
A flag that whether should byte swap.

volatile bool isTxMask
A flag that whether TCR[TXMSK] is set.

volatile uint16_t bytesPerFrame
Number of bytes in each frame

volatile uint16_t frameSize
Backup of TCR[FRAMESZ]

volatile uint8_t fifoSize
FIFO dataSize.

volatile uint8_t rxWatermark
Rx watermark.

volatile uint8_t bytesEachWrite
Bytes for each write TDR.

volatile uint8_t bytesEachRead
Bytes for each read RDR.

const uint8_t *volatile txData
Send buffer.

uint8_t *volatile rxData
Receive buffer.

volatile size_t txRemainingByteCount
Number of bytes remaining to send.

volatile size_t rxRemainingByteCount
Number of bytes remaining to receive.

volatile uint32_t writeRegRemainingTimes
Write TDR register remaining times.

volatile uint32_t readRegRemainingTimes
Read RDR register remaining times.

uint32_t totalByteCount
Number of transfer bytes

uint32_t txBuffIfNull
Used if the txData is NULL.

volatile uint8_t state
LPSPI transfer state , _lpspi_transfer_state.

lpspi_master_transfer_callback_t callback
Completion callback.

void *userData
Callback user data.

struct __lpspi__slave__handle
#include <fsl_lpspi.h> LPSPI slave transfer handle structure used for transactional API.

Public Members

volatile bool isByteSwap
A flag that whether should byte swap.

volatile uint8_t fifoSize
FIFO dataSize.

volatile uint8_t rxWatermark
Rx watermark.

volatile uint8_t bytesEachWrite
Bytes for each write TDR.

volatile uint8_t bytesEachRead
Bytes for each read RDR.

const uint8_t *volatile txData
Send buffer.

uint8_t *volatile rxData
Receive buffer.

volatile size_t txRemainingByteCount
Number of bytes remaining to send.

volatile size_t rxRemainingByteCount
Number of bytes remaining to receive.

volatile uint32_t writeRegRemainingTimes
Write TDR register remaining times.

volatile uint32_t readRegRemainingTimes
Read RDR register remaining times.

uint32_t totalByteCount
Number of transfer bytes

volatile uint8_t state
LPSPI transfer state , _lpspi_transfer_state.

volatile uint32_t errorCount
Error count for slave transfer.

lpspi_slave_transfer_callback_t callback
Completion callback.

void *userData
Callback user data.

2.42 LPSPI eDMA Driver

FSL_LPSPI_EDMA_DRIVER_VERSION
LPSPI EDMA driver version.

DMA_MAX_TRANSFER_COUNT
DMA max transfer size.

typedef struct *_lpspi_master_edma_handle* lpspi_master_edma_handle_t
Forward declaration of the _lpspi_master_edma_handle typedefs.

typedef struct *_lpspi_slave_edma_handle* lpspi_slave_edma_handle_t
Forward declaration of the _lpspi_slave_edma_handle typedefs.

typedef void (*lpspi_master_edma_transfer_callback_t)(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, *status_t* status, void *userData)
Completion callback function pointer type.

Param base
LPSPI peripheral base address.

Param handle
Pointer to the handle for the LPSPI master.

Param status
Success or error code describing whether the transfer completed.

Param userData
Arbitrary pointer-dataSized value passed from the application.

typedef void (*lpspi_slave_edma_transfer_callback_t)(LPSPI_Type *base, *lpspi_slave_edma_handle_t* *handle, *status_t* status, void *userData)
Completion callback function pointer type.

Param base
LPSPI peripheral base address.

Param handle
Pointer to the handle for the LPSPI slave.

Param status
Success or error code describing whether the transfer completed.

Param userData
Arbitrary pointer-dataSized value passed from the application.

void LPSPI_MasterTransferCreateHandleEDMA(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, *lpspi_master_edma_transfer_callback_t* callback, void *userData, *edma_handle_t* *edmaRxRegToRxDataHandle, *edma_handle_t* *edmaTxDataToTxRegHandle)
Initializes the LPSPI master eDMA handle.

This function initializes the LPSPI eDMA handle which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Note that the LPSPI eDMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx are the same source) DMA request source. (1) For a separated DMA request source, enable and set the Rx DMAMUX source for edmaRxRegToRxDataHandle and Tx DMAMUX source for edmaTxDataToTxRegHandle. (2) For a shared DMA request source, enable and set the Rx/Tx DMAMUX source for edmaRxRegToRxDataHandle.

Parameters

- base – LPSPI peripheral base address.
- handle – LPSPI handle pointer to lpspi_master_edma_handle_t.
- callback – LPSPI callback.
- userData – callback function parameter.
- edmaRxRegToRxDataHandle – edmaRxRegToRxDataHandle pointer to edma_handle_t.
- edmaTxDataToTxRegHandle – edmaTxDataToTxRegHandle pointer to edma_handle_t.

status_t LPSPI_MasterTransferEDMA(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI master transfer data using eDMA.

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be an integer multiple of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to lpspi_master_edma_handle_t structure which stores the transfer state.
- transfer – pointer to lpspi_transfer_t structure.

Returns

status of status_t.

status_t LPSPI_MasterTransferPrepareEDMALite(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, uint32_t configFlags)

LPSPI master config transfer parameter while using eDMA.

This function is preparing to transfer data using eDMA, work with LPSPI_MasterTransferEDMALite.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to lpspi_master_edma_handle_t structure which stores the transfer state.
- configFlags – transfer configuration flags. _lpspi_transfer_config_flag_for_master.

Return values

- kStatus_Success – Execution successfully.
- kStatus_LPSPI_Busy – The LPSPI device is busy.

Returns
Indicates whether LPSPI master transfer was successful or not.

status_t LPSPI_MasterTransferEDMALite(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI master transfer data using eDMA without configs.

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: This API is only for transfer through DMA without configuration. Before calling this API, you must call LPSPI_MasterTransferPrepareEDMALite to configure it once. The transfer data size should be an integer multiple of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to lpspi_master_edma_handle_t structure which stores the transfer state.
- transfer – pointer to lpspi_transfer_t structure, config field is not uesed.

Return values

- kStatus_Success – Execution successfully.
- kStatus_LPSPI_Busy – The LPSPI device is busy.
- kStatus_InvalidArgument – The transfer structure is invalid.

Returns
Indicates whether LPSPI master transfer was successful or not.

void LPSPI_MasterTransferAbortEDMA(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle)

LPSPI master aborts a transfer which is using eDMA.

This function aborts a transfer which is using eDMA.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to lpspi_master_edma_handle_t structure which stores the transfer state.

status_t LPSPI_MasterTransferGetCountEDMA(LPSPI_Type *base, *lpspi_master_edma_handle_t* *handle, size_t *count)

Gets the master eDMA transfer remaining bytes.

This function gets the master eDMA transfer remaining bytes.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to lpspi_master_edma_handle_t structure which stores the transfer state.
- count – Number of bytes transferred so far by the EDMA transaction.

Returns
status of status_t.

void LPSPI_SlaveTransferCreateHandleEDMA(LPSPI_Type *base, *lpspi_slave_edma_handle_t* *handle, *lpspi_slave_edma_transfer_callback_t* callback, void *userData, *edma_handle_t* *edmaRxRegToRxDataHandle, *edma_handle_t* *edmaTxDataToTxRegHandle)

Initializes the LPSPI slave eDMA handle.

This function initializes the LPSPI eDMA handle which can be used for other LPSPI transactional APIs. Usually, for a specified LPSPI instance, call this API once to get the initialized handle.

Note that LPSPI eDMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx as the same source) DMA request source.

(1) For a separated DMA request source, enable and set the Rx DMAMUX source for edmaRxRegToRxDataHandle and Tx DMAMUX source for edmaTxDataToTxRegHandle. (2) For a shared DMA request source, enable and set the Rx/Rx DMAMUX source for edmaRxRegToRxDataHandle .

Parameters

- base – LPSPI peripheral base address.
- handle – LPSPI handle pointer to lpspi_slave_edma_handle_t.
- callback – LPSPI callback.
- userData – callback function parameter.
- edmaRxRegToRxDataHandle – edmaRxRegToRxDataHandle pointer to edma_handle_t.
- edmaTxDataToTxRegHandle – edmaTxDataToTxRegHandle pointer to edma_handle_t.

status_t LPSPI_SlaveTransferEDMA(LPSPI_Type *base, *lpspi_slave_edma_handle_t* *handle, *lpspi_transfer_t* *transfer)

LPSPI slave transfers data using eDMA.

This function transfers data using eDMA. This is a non-blocking function, which return right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be an integer multiple of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to lpspi_slave_edma_handle_t structure which stores the transfer state.
- transfer – pointer to lpspi_transfer_t structure.

Returns

status of status_t.

void LPSPI_SlaveTransferAbortEDMA(LPSPI_Type *base, *lpspi_slave_edma_handle_t* *handle)

LPSPI slave aborts a transfer which is using eDMA.

This function aborts a transfer which is using eDMA.

Parameters

- base – LPSPI peripheral base address.

- handle – pointer to lpspi_slave_edma_handle_t structure which stores the transfer state.

status_t LPSPI_SlaveTransferGetCountEDMA(LPSPI_Type *base, *lpspi_slave_edma_handle_t* *handle, size_t *count)

Gets the slave eDMA transfer remaining bytes.

This function gets the slave eDMA transfer remaining bytes.

Parameters

- base – LPSPI peripheral base address.
- handle – pointer to lpspi_slave_edma_handle_t structure which stores the transfer state.
- count – Number of bytes transferred so far by the eDMA transaction.

Returns

status of status_t.

struct __lpspi_master_edma_handle

#include <fsl_lpspi_edma.h> LPSPI master eDMA transfer handle structure used for transactional API.

Public Members

volatile bool isPcsContinuous

Is PCS continuous in transfer.

volatile bool isByteSwap

A flag that whether should byte swap.

volatile uint8_t fifoSize

FIFO dataSize.

volatile uint8_t rxWatermark

Rx watermark.

volatile uint8_t bytesEachWrite

Bytes for each write TDR.

volatile uint8_t bytesEachRead

Bytes for each read RDR.

volatile uint8_t bytesLastRead

Bytes for last read RDR.

volatile bool isThereExtraRxBytes

Is there extra RX byte.

const uint8_t *volatile txData

Send buffer.

uint8_t *volatile rxData

Receive buffer.

volatile size_t txRemainingByteCount

Number of bytes remaining to send.

volatile size_t rxRemainingByteCount

Number of bytes remaining to receive.

volatile uint32_t writeRegRemainingTimes
Write TDR register remaining times.

volatile uint32_t readRegRemainingTimes
Read RDR register remaining times.

uint32_t totalByteCount
Number of transfer bytes

edma_tcd_t *lastTimeTCD
Pointer to the lastTime TCD

bool isMultiDMATransmit
Is there multi DMA transmit

volatile uint8_t dmaTransmitTime
DMA Transfer times.

uint32_t lastTimeDataBytes
DMA transmit last Time data Bytes

uint32_t dataBytesEveryTime
Bytes in a time for DMA transfer, default is DMA_MAX_TRANSFER_COUNT

edma_transfer_config_t transferConfigRx
Config of DMA rx channel.

edma_transfer_config_t transferConfigTx
Config of DMA tx channel.

uint32_t txBuffIfNull
Used if there is not txData for DMA purpose.

uint32_t rxBuffIfNull
Used if there is not rxData for DMA purpose.

uint32_t transmitCommand
Used to write TCR for DMA purpose.

volatile uint8_t state
LPSPI transfer state , _lpspi_transfer_state.

uint8_t nbytes
eDMA minor byte transfer count initially configured.

lpspi_master_edma_transfer_callback_t callback
Completion callback.

void *userData
Callback user data.

edma_handle_t *edmaRxRegToRxDataHandle
edma_handle_t handle point used for RxReg to RxData buff

edma_handle_t *edmaTxDataToTxRegHandle
edma_handle_t handle point used for TxData to TxReg buff

edma_tcd_t lpspiSoftwareTCD[3]
SoftwareTCD, internal used

struct __lpspi_slave_edma_handle
#include <fsl_lpspi_edma.h> LPSPI slave eDMA transfer handle structure used for transactional API.

Public Members

volatile bool isByteSwap
A flag that whether should byte swap.

volatile uint8_t fifoSize
FIFO dataSize.

volatile uint8_t rxWatermark
Rx watermark.

volatile uint8_t bytesEachWrite
Bytes for each write TDR.

volatile uint8_t bytesEachRead
Bytes for each read RDR.

volatile uint8_t bytesLastRead
Bytes for last read RDR.

volatile bool isThereExtraRxBytes
Is there extra RX byte.

uint8_t nbytes
eDMA minor byte transfer count initially configured.

const uint8_t *volatile txData
Send buffer.

uint8_t *volatile rxData
Receive buffer.

volatile size_t txRemainingByteCount
Number of bytes remaining to send.

volatile size_t rxRemainingByteCount
Number of bytes remaining to receive.

volatile uint32_t writeRegRemainingTimes
Write TDR register remaining times.

volatile uint32_t readRegRemainingTimes
Read RDR register remaining times.

uint32_t totalByteCount
Number of transfer bytes

uint32_t txBuffIfNull
Used if there is not txData for DMA purpose.

uint32_t rxBuffIfNull
Used if there is not rxData for DMA purpose.

volatile uint8_t state
LPSPI transfer state.

uint32_t errorCount
Error count for slave transfer.

lpspi_slave_edma_transfer_callback_t callback
Completion callback.

void *userData
Callback user data.

edma_handle_t *edmaRxRegToRxDataHandle
edma_handle_t handle point used for RxReg to RxData buff

edma_handle_t *edmaTxDataToTxRegHandle
edma_handle_t handle point used for TxData to TxReg

edma_tcd_t lpspiSoftwareTCD[2]
SoftwareTCD, internal used

2.43 LPTMR: Low-Power Timer

void LPTMR_Init(LPTMR_Type *base, const *lptmr_config_t* *config)
Ungates the LPTMR clock and configures the peripheral for a basic operation.

Note: This API should be called at the beginning of the application using the LPTMR driver.

Parameters

- base – LPTMR peripheral base address
- config – A pointer to the LPTMR configuration structure.

void LPTMR_Deinit(LPTMR_Type *base)
Gates the LPTMR clock.

Parameters

- base – LPTMR peripheral base address

void LPTMR_GetDefaultConfig(*lptmr_config_t* *config)
Fills in the LPTMR configuration structure with default settings.

The default values are as follows.

```
config->timerMode = kLPTMR_TimerModeTimeCounter;
config->pinSelect = kLPTMR_PinSelectInput_0;
config->pinPolarity = kLPTMR_PinPolarityActiveHigh;
config->enableFreeRunning = false;
config->bypassPrescaler = true;
config->prescalerClockSource = kLPTMR_PrescalerClock_1;
config->value = kLPTMR_Prescale_Glitch_0;
```

Parameters

- config – A pointer to the LPTMR configuration structure.

static inline void LPTMR_EnableInterrupts(LPTMR_Type *base, uint32_t mask)
Enables the selected LPTMR interrupts.

Parameters

- base – LPTMR peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration lptmr_interrupt_enable_t

static inline void LPTMR_DisableInterrupts(LPTMR_Type *base, uint32_t mask)

Disables the selected LPTMR interrupts.

Parameters

- base – LPTMR peripheral base address
- mask – The interrupts to disable. This is a logical OR of members of the enumeration lptmr_interrupt_enable_t.

static inline uint32_t LPTMR_GetEnabledInterrupts(LPTMR_Type *base)

Gets the enabled LPTMR interrupts.

Parameters

- base – LPTMR peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration lptmr_interrupt_enable_t

static inline uint32_t LPTMR_GetStatusFlags(LPTMR_Type *base)

Gets the LPTMR status flags.

Parameters

- base – LPTMR peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration lptmr_status_flags_t

static inline void LPTMR_ClearStatusFlags(LPTMR_Type *base, uint32_t mask)

Clears the LPTMR status flags.

Parameters

- base – LPTMR peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration lptmr_status_flags_t.

static inline void LPTMR_SetTimerPeriod(LPTMR_Type *base, uint32_t ticks)

Sets the timer period in units of count.

Timers counts from 0 until it equals the count value set here. The count value is written to the CMR register.

Note:

a. The TCF flag is set with the CNR equals the count provided here and then increments.

b. Call the utility macros provided in the fsl_common.h to convert to ticks.

Parameters

- base – LPTMR peripheral base address
- ticks – A timer period in units of ticks

static inline uint32_t LPTMR_GetCurrentTimerCount(LPTMR_Type *base)

Reads the current timer counting value.

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note: Call the utility macros provided in the fsl_common.h to convert ticks to usec or msec.

Parameters

- base – LPTMR peripheral base address

Returns
The current counter value in ticks

static inline void LPTMR_StartTimer(LPTMR_Type *base)

Starts the timer.

After calling this function, the timer counts up to the CMR register value. Each time the timer reaches the CMR value and then increments, it generates a trigger pulse and sets the timeout interrupt flag. An interrupt is also triggered if the timer interrupt is enabled.

Parameters

- base – LPTMR peripheral base address

static inline void LPTMR_StopTimer(LPTMR_Type *base)

Stops the timer.

This function stops the timer and resets the timer's counter register.

Parameters

- base – LPTMR peripheral base address

FSL_LPTMR_DRIVER_VERSION

Driver Version

enum _lptmr_pin_select

LPTMR pin selection used in pulse counter mode.

Values:

enumerator kLPTMR_PinSelectInput_0
Pulse counter input 0 is selected

enumerator kLPTMR_PinSelectInput_1
Pulse counter input 1 is selected

enumerator kLPTMR_PinSelectInput_2
Pulse counter input 2 is selected

enumerator kLPTMR_PinSelectInput_3
Pulse counter input 3 is selected

enum _lptmr_pin_polarity

LPTMR pin polarity used in pulse counter mode.

Values:

enumerator kLPTMR_PinPolarityActiveHigh
Pulse Counter input source is active-high

enumerator kLPTMR_PinPolarityActiveLow
Pulse Counter input source is active-low

enum _lptmr_timer_mode

LPTMR timer mode selection.

Values:

enumerator kLPTMR_TimerModeTimeCounter
Time Counter mode

enumerator kLPTMR_TimerModePulseCounter
Pulse Counter mode

enum _lptmr_prescaler_glitch_value
LPTMR prescaler/glitch filter values.

Values:

enumerator kLPTMR_Prescale_Glitch_0
Prescaler divide 2, glitch filter does not support this setting

enumerator kLPTMR_Prescale_Glitch_1
Prescaler divide 4, glitch filter 2

enumerator kLPTMR_Prescale_Glitch_2
Prescaler divide 8, glitch filter 4

enumerator kLPTMR_Prescale_Glitch_3
Prescaler divide 16, glitch filter 8

enumerator kLPTMR_Prescale_Glitch_4
Prescaler divide 32, glitch filter 16

enumerator kLPTMR_Prescale_Glitch_5
Prescaler divide 64, glitch filter 32

enumerator kLPTMR_Prescale_Glitch_6
Prescaler divide 128, glitch filter 64

enumerator kLPTMR_Prescale_Glitch_7
Prescaler divide 256, glitch filter 128

enumerator kLPTMR_Prescale_Glitch_8
Prescaler divide 512, glitch filter 256

enumerator kLPTMR_Prescale_Glitch_9
Prescaler divide 1024, glitch filter 512

enumerator kLPTMR_Prescale_Glitch_10
Prescaler divide 2048 glitch filter 1024

enumerator kLPTMR_Prescale_Glitch_11
Prescaler divide 4096, glitch filter 2048

enumerator kLPTMR_Prescale_Glitch_12
Prescaler divide 8192, glitch filter 4096

enumerator kLPTMR_Prescale_Glitch_13
Prescaler divide 16384, glitch filter 8192

enumerator kLPTMR_Prescale_Glitch_14
Prescaler divide 32768, glitch filter 16384

enumerator kLPTMR_Prescale_Glitch_15
Prescaler divide 65536, glitch filter 32768

enum _lptmr_prescaler_clock_select
LPTMR prescaler/glitch filter clock select.

Note: Clock connections are SoC-specific

Values:

enum _lptmr_interrupt_enable

List of the LPTMR interrupts.

Values:

enumerator kLPTMR_TimerInterruptEnable

Timer interrupt enable

enum _lptmr_status_flags

List of the LPTMR status flags.

Values:

enumerator kLPTMR_TimerCompareFlag

Timer compare flag

typedef enum *_lptmr_pin_select* lptmr_pin_select_t

LPTMR pin selection used in pulse counter mode.

typedef enum *_lptmr_pin_polarity* lptmr_pin_polarity_t

LPTMR pin polarity used in pulse counter mode.

typedef enum *_lptmr_timer_mode* lptmr_timer_mode_t

LPTMR timer mode selection.

typedef enum *_lptmr_prescaler_glitch_value* lptmr_prescaler_glitch_value_t

LPTMR prescaler/glitch filter values.

typedef enum *_lptmr_prescaler_clock_select* lptmr_prescaler_clock_select_t

LPTMR prescaler/glitch filter clock select.

Note: Clock connections are SoC-specific

typedef enum *_lptmr_interrupt_enable* lptmr_interrupt_enable_t

List of the LPTMR interrupts.

typedef enum *_lptmr_status_flags* lptmr_status_flags_t

List of the LPTMR status flags.

typedef struct *_lptmr_config* lptmr_config_t

LPTMR config structure.

This structure holds the configuration settings for the LPTMR peripheral. To initialize this structure to reasonable defaults, call the LPTMR_GetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration struct can be made constant so it resides in flash.

static inline void LPTMR_EnableTimerDMA(LPTMR_Type *base, bool enable)

Enable or disable timer DMA request.

Parameters

- base – base LPTMR peripheral base address
- enable – Switcher of timer DMA feature. "true" means to enable, "false" means to disable.

struct _lptmr_config

#include <fsl_lptmr.h> LPTMR config structure.

This structure holds the configuration settings for the LPTMR peripheral. To initialize this structure to reasonable defaults, call the LPTMR_GetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration struct can be made constant so it resides in flash.

Public Members

lptmr_timer_mode_t timerMode
: Time counter mode or pulse counter mode

lptmr_pin_select_t pinSelect
: LPTMR pulse input pin select; used only in pulse counter mode

lptmr_pin_polarity_t pinPolarity
: LPTMR pulse input pin polarity; used only in pulse counter mode

bool enableFreeRunning
: True: enable free running, counter is reset on overflow False: counter is reset when the compare flag is set

bool bypassPrescaler
: True: bypass prescaler; false: use clock from prescaler

lptmr_prescaler_clock_select_t prescalerClockSource
: LPTMR clock source

lptmr_prescaler_glitch_value_t value
: Prescaler or glitch filter value

2.44 LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver

2.45 LPUART Driver

static inline void LPUART_SoftwareReset(LPUART_Type *base)

Resets the LPUART using software.

This function resets all internal logic and registers except the Global Register. Remains set until cleared by software.

Parameters

- base – LPUART peripheral base address.

status_t LPUART_Init(LPUART_Type *base, const *lpuart_config_t* *config, uint32_t srcClock_Hz)

Initializes an LPUART instance with the user configuration structure and the peripheral clock.

This function configures the LPUART module with user-defined settings. Call the LPUART_GetDefaultConfig() function to configure the configuration structure and get the default configuration. The example below shows how to use this API to configure the LPUART.

```
lpuart_config_t lpuartConfig;
lpuartConfig.baudRate_Bps = 115200U;
lpuartConfig.parityMode = kLPUART_ParityDisabled;
lpuartConfig.dataBitsCount = kLPUART_EightDataBits;
lpuartConfig.isMsb = false;
lpuartConfig.stopBitCount = kLPUART_OneStopBit;
lpuartConfig.txFifoWatermark = 0;
```

(continues on next page)

(continued from previous page)

```
lpuartConfig.rxFifoWatermark = 1;
LPUART_Init(LPUART1, &lpuartConfig, 20000000U);
```

Parameters

- base – LPUART peripheral base address.
- config – Pointer to a user-defined configuration structure.
- srcClock_Hz – LPUART clock source frequency in HZ.

Return values

- kStatus_LPUART_BaudrateNotSupport – Baudrate is not support in current clock source.
- kStatus_Success – LPUART initialize succeed

status_t LPUART_Deinit(LPUART_Type *base)

Deinitializes a LPUART instance.

This function waits for transmit to complete, disables TX and RX, and disables the LPUART clock.

Parameters

- base – LPUART peripheral base address.

Return values

- kStatus_Success – Deinit is success.
- kStatus_LPUART_Timeout – Timeout during deinit.

void LPUART_GetDefaultConfig(*lpuart_config_t* *config)

Gets the default configuration structure.

This function initializes the LPUART configuration structure to a default value. The default values are: lpuartConfig->baudRate_Bps = 115200U; lpuartConfig->parityMode = kLPUART_ParityDisabled; lpuartConfig->dataBitsCount = kLPUART_EightDataBits; lpuartConfig->isMsb = false; lpuartConfig->stopBitCount = kLPUART_OneStopBit; lpuartConfig->txFifoWatermark = 0; lpuartConfig->rxFifoWatermark = 1; lpuartConfig->rxIdleType = kLPUART_IdleTypeStartBit; lpuartConfig->rxIdleConfig = kLPUART_IdleCharacter1; lpuartConfig->enableTx = false; lpuartConfig->enableRx = false;

Parameters

- config – Pointer to a configuration structure.

status_t LPUART_SetBaudRate(LPUART_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)

Sets the LPUART instance baudrate.

This function configures the LPUART module baudrate. This function is used to update the LPUART module baudrate after the LPUART module is initialized by the LPUART_Init.

```
LPUART_SetBaudRate(LPUART1, 115200U, 20000000U);
```

Parameters

- base – LPUART peripheral base address.
- baudRate_Bps – LPUART baudrate to be set.
- srcClock_Hz – LPUART clock source frequency in HZ.

Return values

- kStatus_LPUART_BaudrateNotSupport – Baudrate is not supported in the current clock source.
- kStatus_Success – Set baudrate succeeded.

void LPUART_Enable9bitMode(LPUART_Type *base, bool enable)

Enable 9-bit data mode for LPUART.

This function set the 9-bit mode for LPUART module. The 9th bit is not used for parity thus can be modified by user.

Parameters

- base – LPUART peripheral base address.
- enable – true to enable, flase to disable.

static inline void LPUART_SetMatchAddress(LPUART_Type *base, uint16_t address1, uint16_t address2)

Set the LPUART address.

This function configures the address for LPUART module that works as slave in 9-bit data mode. One or two address fields can be configured. When the address field's match enable bit is set, the frame it receices with MSB being 1 is considered as an address frame, otherwise it is considered as data frame. Once the address frame matches one of slave's own addresses, this slave is addressed. This address frame and its following data frames are stored in the receive buffer, otherwise the frames will be discarded. To un-address a slave, just send an address frame with unmatched address.

Note: Any LPUART instance joined in the multi-slave system can work as slave. The position of the address mark is the same as the parity bit when parity is enabled for 8 bit and 9 bit data formats.

Parameters

- base – LPUART peripheral base address.
- address1 – LPUART slave address1.
- address2 – LPUART slave address2.

static inline void LPUART_EnableMatchAddress(LPUART_Type *base, bool match1, bool match2)

Enable the LPUART match address feature.

Parameters

- base – LPUART peripheral base address.
- match1 – true to enable match address1, false to disable.
- match2 – true to enable match address2, false to disable.

static inline void LPUART_SetRxFifoWatermark(LPUART_Type *base, uint8_t water)

Sets the rx FIFO watermark.

Parameters

- base – LPUART peripheral base address.
- water – Rx FIFO watermark.

static inline void LPUART_SetTxFifoWatermark(LPUART_Type *base, uint8_t water)

Sets the tx FIFO watermark.

Parameters

- base – LPUART peripheral base address.
- water – Tx FIFO watermark.

static inline void LPUART_TransferEnable16Bit(*lpuart_handle_t* *handle, bool enable)

Sets the LPUART using 16bit transmit, only for 9bit or 10bit mode.

This function Enable 16bit Data transmit in lpuart_handle_t.

Parameters

- handle – LPUART handle pointer.
- enable – true to enable, false to disable.

uint32_t LPUART_GetStatusFlags(LPUART_Type *base)

Gets LPUART status flags.

This function gets all LPUART status flags. The flags are returned as the logical OR value of the enumerators _lpuart_flags. To check for a specific status, compare the return value with enumerators in the _lpuart_flags. For example, to check whether the TX is empty:

```
if (kLPUART_TxDataRegEmptyFlag & LPUART_GetStatusFlags(LPUART1))
{
    ...
}
```

Parameters

- base – LPUART peripheral base address.

Returns

LPUART status flags which are ORed by the enumerators in the _lpuart_flags.

status_t LPUART_ClearStatusFlags(LPUART_Type *base, uint32_t mask)

Clears status flags with a provided mask.

This function clears LPUART status flags with a provided mask. Automatically cleared flags can't be cleared by this function. Flags that can only cleared or set by hardware are: kLPUART_TxDataRegEmptyFlag, kLPUART_TransmissionCompleteFlag, kLPUART_RxDataRegFullFlag, kLPUART_RxActiveFlag, kLPUART_NoiseErrorFlag, kLPUART_ParityErrorFlag, kLPUART_TxFifoEmptyFlag,kLPUART_RxFifoEmptyFlag Note: This API should be called when the Tx/Rx is idle, otherwise it takes no effects.

Parameters

- base – LPUART peripheral base address.
- mask – the status flags to be cleared. The user can use the enumerators in the _lpuart_status_flag_t to do the OR operation and get the mask.

Return values

- kStatus_LPUART_FlagCannotClearManually – The flag can't be cleared by this function but it is cleared automatically by hardware.
- kStatus_Success – Status in the mask are cleared.

Returns

0 succeed, others failed.

void LPUART_EnableInterrupts(LPUART_Type *base, uint32_t mask)

Enables LPUART interrupts according to a provided mask.

This function enables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See the _lpuart_interrupt_enable. This examples shows how to enable TX empty interrupt and RX full interrupt:

```
LPUART_EnableInterrupts(LPUART1,kLPUART_TxDataRegEmptyInterruptEnable | kLPUART_
↪RxDataRegFullInterruptEnable);
```

Parameters

- base – LPUART peripheral base address.
- mask – The interrupts to enable. Logical OR of _lpuart_interrupt_enable.

void LPUART_DisableInterrupts(LPUART_Type *base, uint32_t mask)

Disables LPUART interrupts according to a provided mask.

This function disables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See _lpuart_interrupt_enable. This example shows how to disable the TX empty interrupt and RX full interrupt:

```
LPUART_DisableInterrupts(LPUART1,kLPUART_TxDataRegEmptyInterruptEnable | kLPUART_
↪RxDataRegFullInterruptEnable);
```

Parameters

- base – LPUART peripheral base address.
- mask – The interrupts to disable. Logical OR of _lpuart_interrupt_enable.

uint32_t LPUART_GetEnabledInterrupts(LPUART_Type *base)

Gets enabled LPUART interrupts.

This function gets the enabled LPUART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators _lpuart_interrupt_enable. To check a specific interrupt enable status, compare the return value with enumerators in _lpuart_interrupt_enable. For example, to check whether the TX empty interrupt is enabled:

```
uint32_t enabledInterrupts = LPUART_GetEnabledInterrupts(LPUART1);

if (kLPUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)
{
    ...
}
```

Parameters

- base – LPUART peripheral base address.

Returns

LPUART interrupt flags which are logical OR of the enumerators in _lpuart_interrupt_enable.

static inline uintptr_t LPUART_GetDataRegisterAddress(LPUART_Type *base)

Gets the LPUART data register address.

This function returns the LPUART data register address, which is mainly used by the DMA/eDMA.

Parameters

- base – LPUART peripheral base address.

Returns

LPUART data register addresses which are used both by the transmitter and receiver.

static inline void LPUART_EnableTxDMA(LPUART_Type *base, bool enable)

Enables or disables the LPUART transmitter DMA request.

This function enables or disables the transmit data register empty flag, STAT[TDRE], to generate DMA requests.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

static inline void LPUART_EnableRxDMA(LPUART_Type *base, bool enable)

Enables or disables the LPUART receiver DMA.

This function enables or disables the receiver data register full flag, STAT[RDRF], to generate DMA requests.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

uint32_t LPUART_GetInstance(LPUART_Type *base)

Get the LPUART instance from peripheral base address.

Parameters

- base – LPUART peripheral base address.

Returns

LPUART instance.

static inline void LPUART_EnableTx(LPUART_Type *base, bool enable)

Enables or disables the LPUART transmitter.

This function enables or disables the LPUART transmitter.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

static inline void LPUART_EnableRx(LPUART_Type *base, bool enable)

Enables or disables the LPUART receiver.

This function enables or disables the LPUART receiver.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

static inline void LPUART_WriteByte(LPUART_Type *base, uint8_t data)

Writes to the transmitter register.

This function writes data to the transmitter register directly. The upper layer must ensure that the TX register is empty or that the TX FIFO has room before calling this function.

Parameters

- base – LPUART peripheral base address.
- data – Data write to the TX register.

static inline uint8_t LPUART_ReadByte(LPUART_Type *base)

Reads the receiver register.

This function reads data from the receiver register directly. The upper layer must ensure that the receiver register is full or that the RX FIFO has data before calling this function.

Parameters

- base – LPUART peripheral base address.

Returns

Data read from data register.

static inline uint8_t LPUART_GetRxFifoCount(LPUART_Type *base)

Gets the rx FIFO data count.

Parameters

- base – LPUART peripheral base address.

Returns

rx FIFO data count.

static inline uint8_t LPUART_GetTxFifoCount(LPUART_Type *base)

Gets the tx FIFO data count.

Parameters

- base – LPUART peripheral base address.

Returns

tx FIFO data count.

void LPUART_SendAddress(LPUART_Type *base, uint8_t address)

Transmit an address frame in 9-bit data mode.

Parameters

- base – LPUART peripheral base address.
- address – LPUART slave address.

status_t LPUART_WriteBlocking(LPUART_Type *base, const uint8_t *data, size_t length)

Writes to the transmitter register using a blocking method.

This function polls the transmitter register, first waits for the register to be empty or TX FIFO to have room, and writes data to the transmitter buffer, then waits for the dat to be sent out to the bus.

Parameters

- base – LPUART peripheral base address.
- data – Start address of the data to write.
- length – Size of the data to write.

Return values

- kStatus_LPUART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully wrote all data.

status_t LPUART_WriteBlocking16bit(LPUART_Type *base, const uint16_t *data, size_t length)

Writes to the transmitter register using a blocking method in 9bit or 10bit mode.

Note: This function only support 9bit or 10bit transfer. Please make sure only 10bit of data is valid and other bits are 0.

Parameters

- base – LPUART peripheral base address.
- data – Start address of the data to write.
- length – Size of the data to write.

Return values

- kStatus_LPUART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully wrote all data.

status_t LPUART_ReadBlocking(LPUART_Type *base, uint8_t *data, size_t length)

Reads the receiver data register using a blocking method.

This function polls the receiver register, waits for the receiver register full or receiver FIFO has data, and reads data from the TX register.

Parameters

- base – LPUART peripheral base address.
- data – Start address of the buffer to store the received data.
- length – Size of the buffer.

Return values

- kStatus_LPUART_RxHardwareOverrun – Receiver overrun happened while receiving data.
- kStatus_LPUART_NoiseError – Noise error happened while receiving data.
- kStatus_LPUART_FramingError – Framing error happened while receiving data.
- kStatus_LPUART_ParityError – Parity error happened while receiving data.
- kStatus_LPUART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully received all data.

status_t LPUART_ReadBlocking16bit(LPUART_Type *base, uint16_t *data, size_t length)

Reads the receiver data register in 9bit or 10bit mode.

Note: This function only support 9bit or 10bit transfer.

Parameters

- base – LPUART peripheral base address.
- data – Start address of the buffer to store the received data by 16bit, only 10bit is valid.
- length – Size of the buffer.

Return values

- kStatus_LPUART_RxHardwareOverrun – Receiver overrun happened while receiving data.
- kStatus_LPUART_NoiseError – Noise error happened while receiving data.
- kStatus_LPUART_FramingError – Framing error happened while receiving data.

- kStatus_LPUART_ParityError – Parity error happened while receiving data.
- kStatus_LPUART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully received all data.

void LPUART_TransferCreateHandle(LPUART_Type *base, *lpuart_handle_t* *handle, *lpuart_transfer_callback_t* callback, void *userData)

Initializes the LPUART handle.

This function initializes the LPUART handle, which can be used for other LPUART transactional APIs. Usually, for a specified LPUART instance, call this API once to get the initialized handle.

The LPUART driver supports the "background" receiving, which means that user can set up an RX ring buffer optionally. Data received is stored into the ring buffer even when the user doesn't call the LPUART_TransferReceiveNonBlocking() API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly. The ring buffer is disabled if passing NULL as ringBuffer.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- callback – Callback function.
- userData – User data.

status_t LPUART_TransferSendNonBlocking(LPUART_Type *base, *lpuart_handle_t* *handle, *lpuart_transfer_t* *xfer)

Transmits a buffer of data using the interrupt method.

This function send data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data written to the transmitter register. When all data is written to the TX register in the ISR, the LPUART driver calls the callback function and passes the kStatus_LPUART_TxIdle as status parameter.

Note: The kStatus_LPUART_TxIdle is passed to the upper layer when all data are written to the TX register. However, there is no check to ensure that all the data sent out. Before disabling the TX, check the kLPUART_TransmissionCompleteFlag to ensure that the transmit is finished.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- xfer – LPUART transfer structure, see lpuart_transfer_t.

Return values

- kStatus_Success – Successfully start the data transmission.
- kStatus_LPUART_TxBusy – Previous transmission still not finished, data not all written to the TX register.
- kStatus_InvalidArgument – Invalid argument.

void LPUART_TransferStartRingBuffer(LPUART_Type *base, *lpuart_handle_t* *handle, uint8_t *ringBuffer, size_t ringBufferSize)

Sets up the RX ring buffer.

This function sets up the RX ring buffer to a specific UART handle.

When the RX ring buffer is used, data received is stored into the ring buffer even when the user doesn't call the UART_TransferReceiveNonBlocking() API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly.

Note: When using RX ring buffer, one byte is reserved for internal use. In other words, if ringBufferSize is 32, then only 31 bytes are used for saving data.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- ringBuffer – Start address of ring buffer for background receiving. Pass NULL to disable the ring buffer.
- ringBufferSize – size of the ring buffer.

void LPUART_TransferStopRingBuffer(LPUART_Type *base, *lpuart_handle_t* *handle)

Aborts the background transfer and uninstalls the ring buffer.

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.

size_t LPUART_TransferGetRxRingBufferLength(LPUART_Type *base, *lpuart_handle_t* *handle)

Get the length of received data in RX ring buffer.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.

Returns

Length of received data in RX ring buffer.

void LPUART_TransferAbortSend(LPUART_Type *base, *lpuart_handle_t* *handle)

Aborts the interrupt-driven data transmit.

This function aborts the interrupt driven data sending. The user can get the remainBtyes to find out how many bytes are not sent out.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.

status_t LPUART_TransferGetSendCount(LPUART_Type *base, *lpuart_handle_t* *handle, uint32_t *count)

Gets the number of bytes that have been sent out to bus.

This function gets the number of bytes that have been sent out to bus by an interrupt method.

Parameters

- base – LPUART peripheral base address.

- handle – LPUART handle pointer.
- count – Send bytes count.

Return values

- kStatus_NoTransferInProgress – No send in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

status_t LPUART_TransferReceiveNonBlocking(LPUART_Type *base, *lpuart_handle_t* *handle, *lpuart_transfer_t* *xfer, size_t *receivedBytes)

Receives a buffer of data using the interrupt method.

This function receives data using an interrupt method. This is a non-blocking function which returns without waiting to ensure that all data are received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter receivedBytes shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is not enough for read, the receive request is saved by the LPUART driver. When the new data arrives, the receive request is serviced first. When all data is received, the LPUART driver notifies the upper layer through a callback function and passes a status parameter kStatus_UART_RxIdle. For example, the upper layer needs 10 bytes but there are only 5 bytes in ring buffer. The 5 bytes are copied to xfer->data, which returns with the parameter receivedBytes set to 5. For the remaining 5 bytes, the newly arrived data is saved from xfer->data[5]. When 5 bytes are received, the LPUART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to xfer->data. When all data is received, the upper layer is notified.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- xfer – LPUART transfer structure, see uart_transfer_t.
- receivedBytes – Bytes received from the ring buffer directly.

Return values

- kStatus_Success – Successfully queue the transfer into the transmit queue.
- kStatus_LPUART_RxBusy – Previous receive request is not finished.
- kStatus_InvalidArgument – Invalid argument.

void LPUART_TransferAbortReceive(LPUART_Type *base, *lpuart_handle_t* *handle)

Aborts the interrupt-driven data receiving.

This function aborts the interrupt-driven data receiving. The user can get the remainBytes to find out how many bytes not received yet.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.

status_t LPUART_TransferGetReceiveCount(LPUART_Type *base, *lpuart_handle_t* *handle, uint32_t *count)

Gets the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- base – LPUART peripheral base address.

- handle – LPUART handle pointer.
- count – Receive bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

void LPUART_TransferHandleIRQ(LPUART_Type *base, void *irqHandle)

LPUART IRQ handle function.

This function handles the LPUART transmit and receive IRQ request.

Parameters

- base – LPUART peripheral base address.
- irqHandle – LPUART handle pointer.

void LPUART_TransferHandleErrorIRQ(LPUART_Type *base, void *irqHandle)

LPUART Error IRQ handle function.

This function handles the LPUART error IRQ request.

Parameters

- base – LPUART peripheral base address.
- irqHandle – LPUART handle pointer.

void LPUART_DriverIRQHandler(uint32_t instance)

LPUART driver IRQ handler common entry.

This function provides the common IRQ request entry for LPUART.

Parameters

- instance – LPUART instance.

FSL_LPUART_DRIVER_VERSION

LPUART driver version.

Error codes for the LPUART driver.

Values:

enumerator kStatus_LPUART_TxBusy

TX busy

enumerator kStatus_LPUART_RxBusy

RX busy

enumerator kStatus_LPUART_TxIdle

LPUART transmitter is idle.

enumerator kStatus_LPUART_RxIdle

LPUART receiver is idle.

enumerator kStatus_LPUART_TxWatermarkTooLarge

TX FIFO watermark too large

enumerator kStatus_LPUART_RxWatermarkTooLarge

RX FIFO watermark too large

enumerator kStatus_LPUART_FlagCannotClearManually
Some flag can't manually clear

enumerator kStatus_LPUART_Error
Error happens on LPUART.

enumerator kStatus_LPUART_RxRingBufferOverrun
LPUART RX software ring buffer overrun.

enumerator kStatus_LPUART_RxHardwareOverrun
LPUART RX receiver overrun.

enumerator kStatus_LPUART_NoiseError
LPUART noise error.

enumerator kStatus_LPUART_FramingError
LPUART framing error.

enumerator kStatus_LPUART_ParityError
LPUART parity error.

enumerator kStatus_LPUART_BaudrateNotSupport
Baudrate is not support in current clock source

enumerator kStatus_LPUART_IdleLineDetected
IDLE flag.

enumerator kStatus_LPUART_Timeout
LPUART times out.

enum _lpuart_parity_mode
LPUART parity mode.

Values:

enumerator kLPUART_ParityDisabled
Parity disabled

enumerator kLPUART_ParityEven
Parity enabled, type even, bit setting: PE|PT = 10

enumerator kLPUART_ParityOdd
Parity enabled, type odd, bit setting: PE|PT = 11

enum _lpuart_data_bits
LPUART data bits count.

Values:

enumerator kLPUART_EightDataBits
Eight data bit

enumerator kLPUART_SevenDataBits
Seven data bit

enum _lpuart_stop_bit_count
LPUART stop bit count.

Values:

enumerator kLPUART_OneStopBit
One stop bit

enumerator kLPUART_TwoStopBit
Two stop bits

enum _lpuart_transmit_cts_source
LPUART transmit CTS source.

Values:

enumerator kLPUART_CtsSourcePin
CTS resource is the LPUART_CTS pin.

enumerator kLPUART_CtsSourceMatchResult
CTS resource is the match result.

enum _lpuart_transmit_cts_config
LPUART transmit CTS configure.

Values:

enumerator kLPUART_CtsSampleAtStart
CTS input is sampled at the start of each character.

enumerator kLPUART_CtsSampleAtIdle
CTS input is sampled when the transmitter is idle

enum _lpuart_idle_type_select
LPUART idle flag type defines when the receiver starts counting.

Values:

enumerator kLPUART_IdleTypeStartBit
Start counting after a valid start bit.

enumerator kLPUART_IdleTypeStopBit
Start counting after a stop bit.

enum _lpuart_idle_config
LPUART idle detected configuration. This structure defines the number of idle characters that must be received before the IDLE flag is set.

Values:

enumerator kLPUART_IdleCharacter1
the number of idle characters.

enumerator kLPUART_IdleCharacter2
the number of idle characters.

enumerator kLPUART_IdleCharacter4
the number of idle characters.

enumerator kLPUART_IdleCharacter8
the number of idle characters.

enumerator kLPUART_IdleCharacter16
the number of idle characters.

enumerator kLPUART_IdleCharacter32
the number of idle characters.

enumerator kLPUART_IdleCharacter64
the number of idle characters.

enumerator kLPUART_IdleCharacter128
the number of idle characters.

enum _lpuart_interrupt_enable
LPUART interrupt configuration structure, default settings all disabled.

This structure contains the settings for all LPUART interrupt configurations.

Values:

enumerator kLPUART_LinBreakInterruptEnable
LIN break detect. bit 7

enumerator kLPUART_RxActiveEdgeInterruptEnable
Receive Active Edge. bit 6

enumerator kLPUART_TxDataRegEmptyInterruptEnable
Transmit data register empty. bit 23

enumerator kLPUART_TransmissionCompleteInterruptEnable
Transmission complete. bit 22

enumerator kLPUART_RxDataRegFullInterruptEnable
Receiver data register full. bit 21

enumerator kLPUART_IdleLineInterruptEnable
Idle line. bit 20

enumerator kLPUART_RxOverrunInterruptEnable
Receiver Overrun. bit 27

enumerator kLPUART_NoiseErrorInterruptEnable
Noise error flag. bit 26

enumerator kLPUART_FramingErrorInterruptEnable
Framing error flag. bit 25

enumerator kLPUART_ParityErrorInterruptEnable
Parity error flag. bit 24

enumerator kLPUART_Match1InterruptEnable
Parity error flag. bit 15

enumerator kLPUART_Match2InterruptEnable
Parity error flag. bit 14

enumerator kLPUART_TxFifoOverflowInterruptEnable
Transmit FIFO Overflow. bit 9

enumerator kLPUART_RxFifoUnderflowInterruptEnable
Receive FIFO Underflow. bit 8

enumerator kLPUART_AllInterruptEnable

enum _lpuart_flags
LPUART status flags.

This provides constants for the LPUART status flags for use in the LPUART functions.

Values:

enumerator kLPUART_TxDataRegEmptyFlag
Transmit data register empty flag, sets when transmit buffer is empty. bit 23

enumerator kLPUART_TransmissionCompleteFlag
Transmission complete flag, sets when transmission activity complete. bit 22

enumerator kLPUART_RxDataRegFullFlag
Receive data register full flag, sets when the receive data buffer is full. bit 21

enumerator kLPUART_IdleLineFlag
Idle line detect flag, sets when idle line detected. bit 20

enumerator kLPUART_RxOverrunFlag
Receive Overrun, sets when new data is received before data is read from receive register. bit 19

enumerator kLPUART_NoiseErrorFlag
Receive takes 3 samples of each received bit. If any of these samples differ, noise flag sets. bit 18

enumerator kLPUART_FramingErrorFlag
Frame error flag, sets if logic 0 was detected where stop bit expected. bit 17

enumerator kLPUART_ParityErrorFlag
If parity enabled, sets upon parity error detection. bit 16

enumerator kLPUART_LinBreakFlag
LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled. bit 31

enumerator kLPUART_RxActiveEdgeFlag
Receive pin active edge interrupt flag, sets when active edge detected. bit 30

enumerator kLPUART_RxActiveFlag
Receiver Active Flag (RAF), sets at beginning of valid start. bit 24

enumerator kLPUART_DataMatch1Flag
The next character to be read from LPUART_DATA matches MA1. bit 15

enumerator kLPUART_DataMatch2Flag
The next character to be read from LPUART_DATA matches MA2. bit 14

enumerator kLPUART_TxFifoEmptyFlag
TXEMPT bit, sets if transmit buffer is empty. bit 7

enumerator kLPUART_RxFifoEmptyFlag
RXEMPT bit, sets if receive buffer is empty. bit 6

enumerator kLPUART_TxFifoOverflowFlag
TXOF bit, sets if transmit buffer overflow occurred. bit 1

enumerator kLPUART_RxFifoUnderflowFlag
RXUF bit, sets if receive buffer underflow occurred. bit 0

enumerator kLPUART_AllClearFlags

enumerator kLPUART_AllFlags

typedef enum *_lpuart_parity_mode* lpuart_parity_mode_t
LPUART parity mode.

typedef enum *_lpuart_data_bits* lpuart_data_bits_t
LPUART data bits count.

typedef enum *_lpuart_stop_bit_count* lpuart_stop_bit_count_t
LPUART stop bit count.

typedef enum *_lpuart_transmit_cts_source* lpuart_transmit_cts_source_t

LPUART transmit CTS source.

typedef enum *_lpuart_transmit_cts_config* lpuart_transmit_cts_config_t

LPUART transmit CTS configure.

typedef enum *_lpuart_idle_type_select* lpuart_idle_type_select_t

LPUART idle flag type defines when the receiver starts counting.

typedef enum *_lpuart_idle_config* lpuart_idle_config_t

LPUART idle detected configuration. This structure defines the number of idle characters that must be received before the IDLE flag is set.

typedef struct *_lpuart_config* lpuart_config_t

LPUART configuration structure.

typedef struct *_lpuart_transfer* lpuart_transfer_t

LPUART transfer structure.

typedef struct *_lpuart_handle* lpuart_handle_t

typedef void (*lpuart_transfer_callback_t)(LPUART_Type *base, *lpuart_handle_t* *handle, *status_t* status, void *userData)

LPUART transfer callback function.

typedef void (*lpuart_isr_t)(LPUART_Type *base, void *handle)

void *s_lpuartHandle[]

const IRQn_Type s_lpuartTxIRQ[]

lpuart_isr_t s_lpuartIsr[]

UART_RETRY_TIMES

Retry times for waiting flag.

struct _lpuart_config

#include <fsl_lpuart.h> LPUART configuration structure.

Public Members

uint32_t baudRate_Bps

LPUART baud rate

lpuart_parity_mode_t parityMode

Parity mode, disabled (default), even, odd

lpuart_data_bits_t dataBitsCount

Data bits count, eight (default), seven

bool isMsb

Data bits order, LSB (default), MSB

lpuart_stop_bit_count_t stopBitCount

Number of stop bits, 1 stop bit (default) or 2 stop bits

uint8_t txFifoWatermark

TX FIFO watermark

uint8_t rxFifoWatermark

RX FIFO watermark

bool enableRxRTS
RX RTS enable

bool enableTxCTS
TX CTS enable

lpuart_transmit_cts_source_t txCtsSource
TX CTS source

lpuart_transmit_cts_config_t txCtsConfig
TX CTS configure

uint8_t rtsWatermark
RTS watermark

lpuart_idle_type_select_t rxIdleType
RX IDLE type.

lpuart_idle_config_t rxIdleConfig
RX IDLE configuration.

bool enableTx
Enable TX

bool enableRx
Enable RX

bool swapTxdRxd
Swap TXD and RXD pins

struct __lpuart_transfer
#include <fsl_lpuart.h> LPUART transfer structure.

Public Members

size_t dataSize
The byte count to be transfer.

struct __lpuart_handle
#include <fsl_lpuart.h> LPUART handle structure.

Public Members

volatile size_t txDataSize
Size of the remaining data to send.

size_t txDataSizeAll
Size of the data to send out.

volatile size_t rxDataSize
Size of the remaining data to receive.

size_t rxDataSizeAll
Size of the data to receive.

size_t rxRingBufferSize
Size of the ring buffer.

volatile uint16_t rxRingBufferHead
Index for the driver to store received data into ring buffer.

volatile uint16_t rxRingBufferTail
Index for the user to get data from the ring buffer.

lpuart_transfer_callback_t callback
Callback function.

void *userData
LPUART callback function parameter.

volatile uint8_t txState
TX transfer state.

volatile uint8_t rxState
RX transfer state.

bool isSevenDataBits
Seven data bits flag.

bool is16bitData
16bit data bits flag, only used for 9bit or 10bit data

union ___unnamed58___

Public Members

uint8_t *data
The buffer of data to be transfer.

uint8_t *rxData
The buffer to receive data.

uint16_t *rxData16
The buffer to receive data.

const uint8_t *txData
The buffer of data to be sent.

const uint16_t *txData16
The buffer of data to be sent.

union ___unnamed60___

Public Members

const uint8_t *volatile txData
Address of remaining data to send.

const uint16_t *volatile txData16
Address of remaining data to send.

union ___unnamed62___

Public Members

uint8_t *volatile rxData
Address of remaining data to receive.

uint16_t *volatile rxData16
Address of remaining data to receive.

union ___unnamed64___

Public Members

uint8_t *rxRingBuffer
Start address of the receiver ring buffer.

uint16_t *rxRingBuffer16
Start address of the receiver ring buffer.

2.46 LPUART eDMA Driver

void LPUART_TransferCreateHandleEDMA(LPUART_Type *base, *lpuart_edma_handle_t* *handle, *lpuart_edma_transfer_callback_t* callback, void *userData, *edma_handle_t* *txEdmaHandle, *edma_handle_t* *rxEdmaHandle)

Initializes the LPUART handle which is used in transactional functions.

Note: This function disables all LPUART interrupts.

Parameters

- base – LPUART peripheral base address.
- handle – Pointer to lpuart_edma_handle_t structure.
- callback – Callback function.
- userData – User data.
- txEdmaHandle – User requested DMA handle for TX DMA transfer.
- rxEdmaHandle – User requested DMA handle for RX DMA transfer.

status_t LPUART_SendEDMA(LPUART_Type *base, *lpuart_edma_handle_t* *handle, *lpuart_transfer_t* *xfer)

Sends data using eDMA.

This function sends data using eDMA. This is a non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- xfer – LPUART eDMA transfer structure. See lpuart_transfer_t.

Return values

- kStatus_Success – if succeed, others failed.
- kStatus_LPUART_TxBusy – Previous transfer on going.
- kStatus_InvalidArgument – Invalid argument.

status_t LPUART_ReceiveEDMA(LPUART_Type *base, *lpuart_edma_handle_t* *handle, *lpuart_transfer_t* *xfer)

Receives data using eDMA.

This function receives data using eDMA. This is non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

- base – LPUART peripheral base address.
- handle – Pointer to lpuart_edma_handle_t structure.
- xfer – LPUART eDMA transfer structure, see lpuart_transfer_t.

Return values

- kStatus_Success – if succeed, others fail.
- kStatus_LPUART_RxBusy – Previous transfer ongoing.
- kStatus_InvalidArgument – Invalid argument.

void LPUART_TransferAbortSendEDMA(LPUART_Type *base, *lpuart_edma_handle_t* *handle)

Aborts the sent data using eDMA.

This function aborts the sent data using eDMA.

Parameters

- base – LPUART peripheral base address.
- handle – Pointer to lpuart_edma_handle_t structure.

void LPUART_TransferAbortReceiveEDMA(LPUART_Type *base, *lpuart_edma_handle_t* *handle)

Aborts the received data using eDMA.

This function aborts the received data using eDMA.

Parameters

- base – LPUART peripheral base address.
- handle – Pointer to lpuart_edma_handle_t structure.

status_t LPUART_TransferGetSendCountEDMA(LPUART_Type *base, *lpuart_edma_handle_t* *handle, uint32_t *count)

Gets the number of bytes written to the LPUART TX register.

This function gets the number of bytes written to the LPUART TX register by DMA.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- count – Send bytes count.

Return values

- kStatus_NoTransferInProgress – No send in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

status_t LPUART_TransferGetReceiveCountEDMA(LPUART_Type *base, *lpuart_edma_handle_t* *handle, uint32_t *count)

Gets the number of received bytes.

This function gets the number of received bytes.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- count – Receive bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

void LPUART_TransferEdmaHandleIRQ(LPUART_Type *base, void *lpuartEdmaHandle)

LPUART eDMA IRQ handle function.

This function handles the LPUART tx complete IRQ request and invoke user callback. It is not set to static so that it can be used in user application.

Note: This function is used as default IRQ handler by double weak mechanism. If user's specific IRQ handler is implemented, make sure this function is invoked in the handler.

Parameters

- base – LPUART peripheral base address.
- lpuartEdmaHandle – LPUART handle pointer.

FSL_LPUART_EDMA_DRIVER_VERSION

LPUART EDMA driver version.

typedef struct *_lpuart_edma_handle* lpuart_edma_handle_t

typedef void (*lpuart_edma_transfer_callback_t)(LPUART_Type *base, *lpuart_edma_handle_t* *handle, *status_t* status, void *userData)

LPUART transfer callback function.

struct _lpuart_edma_handle

#include <fsl_lpuart_edma.h> LPUART eDMA handle.

Public Members

lpuart_edma_transfer_callback_t callback

Callback function.

void *userData

LPUART callback function parameter.

size_t rxDataSizeAll

Size of the data to receive.

size_t txDataSizeAll

Size of the data to send out.

edma_handle_t *txEdmaHandle

The eDMA TX channel used.

edma_handle_t *rxEdmaHandle

The eDMA RX channel used.

uint8_t nbytes

eDMA minor byte transfer count initially configured.

volatile uint8_t txState

TX transfer state.

volatile uint8_t rxState

RX transfer state

2.47 LTC: LP Trusted Cryptography

FSL_LTC_DRIVER_VERSION

LTC driver version. Version 2.0.18.

Current version: 2.0.18

Change log:

- Version 2.0.1
  - fixed warning during g++ compilation
- Version 2.0.2
  - fixed [KPSDK-10932][LTC][SHA] LTC_HASH() blocks indefinitely when message size exceeds 4080 bytes
- Version 2.0.3
  - fixed LTC_PKHA_CompareBigNum() in case an integer argument is an array of all zeroes
- Version 2.0.4
  - constant LTC_PKHA_CompareBigNum() processing time
- Version 2.0.5
  - Fix MISRA issues
- Version 2.0.6
  - fixed [KPSDK-23603][LTC] AES Decrypt in ECB and CBC modes fail when ciphertext size > 0xff0 bytes
- Version 2.0.7
  - Fix MISRA-2012 issues
- Version 2.0.8
  - Fix Coverity issues
- Version 2.0.9
  - Fix sign-compare warning in ltc_set_context and in ltc_get_context
- Version 2.0.10
  - Fix MISRA-2012 issues
- Version 2.0.11
  - Fix MISRA-2012 issues
- Version 2.0.12
  - Fix AES Decrypt in CBC modes fail when used kLTC_DecryptKey.
- Version 2.0.13
  - Add feature macro FSL_FEATURE_LTC_HAS_NO_CLOCK_CONTROL_BIT into LTC_Init function.
- Version 2.0.14
  - Add feature macro FSL_FEATURE_LTC_HAS_NO_CLOCK_CONTROL_BIT into LTC_Deinit function.
- Version 2.0.15
  - Fix MISRA-2012 issues

- Version 2.0.16
  - Fix unitialized GCC warning in LTC_AES_GenerateDecryptKey()
- Version 2.0.17
  - Fix CMAC for payloads over one block, and if BRIC is present on the device, remove XCBC and "decrypt key" functionality
- Version 2.0.18
  - Fix CERT INT30-C and INT31-C compliance

void LTC_Init(LTC_Type *base)

Initializes the LTC driver. This function initializes the LTC driver.

Parameters

- base – LTC peripheral base address

void LTC_Deinit(LTC_Type *base)

Deinitializes the LTC driver. This function deinitializes the LTC driver.

Parameters

- base – LTC peripheral base address

void LTC_SetDpaMaskSeed(LTC_Type *base, uint32_t mask)

Sets the DPA Mask Seed register.

The DPA Mask Seed register reseeds the mask that provides resistance against DPA (differential power analysis) attacks on AES or DES keys.

Differential Power Analysis Mask (DPA) resistance uses a randomly changing mask that introduces "noise" into the power consumed by the AES or DES. This reduces the signal-to-noise ratio that differential power analysis attacks use to "guess" bits of the key. This randomly changing mask should be seeded at POR, and continues to provide DPA resistance from that point on. However, to provide even more DPA protection it is recommended that the DPA mask be reseeded after every 50,000 blocks have been processed. At that time, software can opt to write a new seed (preferably obtained from an RNG) into the DPA Mask Seed register (DPAMS), or software can opt to provide the new seed earlier or later, or not at all. DPA resistance continues even if the DPA mask is never reseeded.

Parameters

- base – LTC peripheral base address
- mask – The DPA mask seed.

2.48 LTC AES driver

enum _ltc_aes_key_t

Type of AES key for ECB and CBC decrypt operations.

Values:

enumerator kLTC_EncryptKey

Input key is an encrypt key

typedef enum *_ltc_aes_key_t* ltc_aes_key_t

Type of AES key for ECB and CBC decrypt operations.

status_t LTC_AES_EncryptEcb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t *key, uint32_t keySize)

Encrypts AES using the ECB block mode.

Encrypts AES using the ECB block mode.

Parameters

- base – LTC peripheral base address
- plaintext – Input plain text to encrypt
- ciphertext – **[out]** Output cipher text
- size – Size of input and output data in bytes. Must be multiple of 16 bytes.
- key – Input key to use for encryption
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.

Returns

Status from encrypt operation

status_t LTC_AES_DecryptEcb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t *key, uint32_t keySize, *ltc_aes_key_t* keyType)

Decrypts AES using ECB block mode.

Decrypts AES using ECB block mode.

Parameters

- base – LTC peripheral base address
- ciphertext – Input cipher text to decrypt
- plaintext – **[out]** Output plain text
- size – Size of input and output data in bytes. Must be multiple of 16 bytes.
- key – Input key.
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.
- keyType – Input type of the key (allows to directly load decrypt key for AES ECB decrypt operation.)

Returns

Status from decrypt operation

status_t LTC_AES_EncryptCbc(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[16], const uint8_t *key, uint32_t keySize)

Encrypts AES using CBC block mode.

Parameters

- base – LTC peripheral base address
- plaintext – Input plain text to encrypt
- ciphertext – **[out]** Output cipher text
- size – Size of input and output data in bytes. Must be multiple of 16 bytes.
- iv – Input initial vector to combine with the first input block.
- key – Input key to use for encryption
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.

Returns
Status from encrypt operation

status_t LTC_AES_DecryptCbc(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[16], const uint8_t *key, uint32_t keySize, *ltc_aes_key_t* keyType)

Decrypts AES using CBC block mode.

Parameters

- base – LTC peripheral base address
- ciphertext – Input cipher text to decrypt
- plaintext – **[out]** Output plain text
- size – Size of input and output data in bytes. Must be multiple of 16 bytes.
- iv – Input initial vector to combine with the first input block.
- key – Input key to use for decryption
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.
- keyType – Input type of the key (allows to directly load decrypt key for AES CBC decrypt operation.)

Returns
Status from decrypt operation

status_t LTC_AES_CryptCtr(LTC_Type *base, const uint8_t *input, uint8_t *output, uint32_t size, uint8_t counter[16U], const uint8_t *key, uint32_t keySize, uint8_t counterlast[16U], uint32_t *szLeft)

Encrypts or decrypts AES using CTR block mode.

Encrypts or decrypts AES using CTR block mode. AES CTR mode uses only forward AES cipher and same algorithm for encryption and decryption. The only difference between encryption and decryption is that, for encryption, the input argument is plain text and the output argument is cipher text. For decryption, the input argument is cipher text and the output argument is plain text.

Parameters

- base – LTC peripheral base address
- input – Input data for CTR block mode
- output – **[out]** Output data for CTR block mode
- size – Size of input and output data in bytes
- counter – **[inout]** Input counter (updates on return)
- key – Input key to use for forward AES cipher
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.
- counterlast – **[out]** Output cipher of last counter, for chained CTR calls. NULL can be passed if chained calls are not used.
- szLeft – **[out]** Output number of bytes in left unused in counterlast block. NULL can be passed if chained calls are not used.

Returns
Status from encrypt operation

status_t LTC_AES_EncryptTagGcm(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t *iv, uint32_t ivSize, const uint8_t *aad, uint32_t aadSize, const uint8_t *key, uint32_t keySize, uint8_t *tag, uint32_t tagSize)

Encrypts AES and tags using GCM block mode.

Encrypts AES and optionally tags using GCM block mode. If plaintext is NULL, only the GHASH is calculated and output in the 'tag' field.

Parameters

- base – LTC peripheral base address
- plaintext – Input plain text to encrypt
- ciphertext – **[out]** Output cipher text.
- size – Size of input and output data in bytes
- iv – Input initial vector
- ivSize – Size of the IV
- aad – Input additional authentication data
- aadSize – Input size in bytes of AAD
- key – Input key to use for encryption
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.
- tag – **[out]** Output hash tag. Set to NULL to skip tag processing.
- tagSize – Input size of the tag to generate, in bytes. Must be 4,8,12,13,14,15 or 16.

Returns

Status from encrypt operation

status_t LTC_AES_DecryptTagGcm(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t *iv, uint32_t ivSize, const uint8_t *aad, uint32_t aadSize, const uint8_t *key, uint32_t keySize, const uint8_t *tag, uint32_t tagSize)

Decrypts AES and authenticates using GCM block mode.

Decrypts AES and optionally authenticates using GCM block mode. If ciphertext is NULL, only the GHASH is calculated and compared with the received GHASH in 'tag' field.

Parameters

- base – LTC peripheral base address
- ciphertext – Input cipher text to decrypt
- plaintext – **[out]** Output plain text.
- size – Size of input and output data in bytes
- iv – Input initial vector
- ivSize – Size of the IV
- aad – Input additional authentication data
- aadSize – Input size in bytes of AAD
- key – Input key to use for encryption
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.
- tag – Input hash tag to compare. Set to NULL to skip tag processing.
- tagSize – Input size of the tag, in bytes. Must be 4, 8, 12, 13, 14, 15, or 16.

Returns

Status from decrypt operation

status_t LTC_AES_EncryptTagCcm(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t *iv, uint32_t ivSize, const uint8_t *aad, uint32_t aadSize, const uint8_t *key, uint32_t keySize, uint8_t *tag, uint32_t tagSize)

Encrypts AES and tags using CCM block mode.

Encrypts AES and optionally tags using CCM block mode.

Parameters

- base – LTC peripheral base address
- plaintext – Input plain text to encrypt
- ciphertext – **[out]** Output cipher text.
- size – Size of input and output data in bytes. Zero means authentication only.
- iv – Nonce
- ivSize – Length of the Nonce in bytes. Must be 7, 8, 9, 10, 11, 12, or 13.
- aad – Input additional authentication data. Can be NULL if aadSize is zero.
- aadSize – Input size in bytes of AAD. Zero means data mode only (authentication skipped).
- key – Input key to use for encryption
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.
- tag – **[out]** Generated output tag. Set to NULL to skip tag processing.
- tagSize – Input size of the tag to generate, in bytes. Must be 4, 6, 8, 10, 12, 14, or 16.

Returns

Status from encrypt operation

status_t LTC_AES_DecryptTagCcm(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t *iv, uint32_t ivSize, const uint8_t *aad, uint32_t aadSize, const uint8_t *key, uint32_t keySize, const uint8_t *tag, uint32_t tagSize)

Decrypts AES and authenticates using CCM block mode.

Decrypts AES and optionally authenticates using CCM block mode.

Parameters

- base – LTC peripheral base address
- ciphertext – Input cipher text to decrypt
- plaintext – **[out]** Output plain text.
- size – Size of input and output data in bytes. Zero means authentication only.
- iv – Nonce
- ivSize – Length of the Nonce in bytes. Must be 7, 8, 9, 10, 11, 12, or 13.
- aad – Input additional authentication data. Can be NULL if aadSize is zero.
- aadSize – Input size in bytes of AAD. Zero means data mode only (authentication skipped).
- key – Input key to use for decryption
- keySize – Size of the input key, in bytes. Must be 16, 24, or 32.

- tag – Received tag. Set to NULL to skip tag processing.
- tagSize – Input size of the received tag to compare with the computed tag, in bytes. Must be 4, 6, 8, 10, 12, 14, or 16.

Returns
Status from decrypt operation

LTC_AES_BLOCK_SIZE
AES block size in bytes

LTC_AES_IV_SIZE
AES Input Vector size in bytes

LTC_KEY_REGISTER_READABLE

LTC_AES_DecryptCtr(base, input, output, size, counter, key, keySize, counterlast, szLeft)
AES CTR decrypt is mapped to the AES CTR generic operation

LTC_AES_EncryptCtr(base, input, output, size, counter, key, keySize, counterlast, szLeft)
AES CTR encrypt is mapped to the AES CTR generic operation

2.49 LTC DES driver

status_t LTC_DES_EncryptEcb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t key[8])

Encrypts DES using ECB block mode.

Encrypts DES using ECB block mode.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes. Must be multiple of 8 bytes.
- key – Input key to use for encryption

Returns
Status from encrypt/decrypt operation

status_t LTC_DES_DecryptEcb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t key[8])

Decrypts DES using ECB block mode.

Decrypts DES using ECB block mode.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes. Must be multiple of 8 bytes.
- key – Input key to use for decryption

Returns
Status from encrypt/decrypt operation

status_t LTC_DES_EncryptCbc(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[8], const uint8_t key[8])

Encrypts DES using CBC block mode.

Encrypts DES using CBC block mode.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Ouput ciphertext
- size – Size of input and output data in bytes
- iv – Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
- key – Input key to use for encryption

Returns

Status from encrypt/decrypt operation

status_t LTC_DES_DecryptCbc(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[8], const uint8_t key[8])

Decrypts DES using CBC block mode.

Decrypts DES using CBC block mode.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input data in bytes
- iv – Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
- key – Input key to use for decryption

Returns

Status from encrypt/decrypt operation

status_t LTC_DES_EncryptCfb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[8], const uint8_t key[8])

Encrypts DES using CFB block mode.

Encrypts DES using CFB block mode.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- size – Size of input data in bytes
- iv – Input initial block.
- key – Input key to use for encryption
- ciphertext – **[out]** Output ciphertext

Returns

Status from encrypt/decrypt operation

status_t LTC_DES_DecryptCfb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[8], const uint8_t key[8])

Decrypts DES using CFB block mode.

Decrypts DES using CFB block mode.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes
- iv – Input initial block.
- key – Input key to use for decryption

Returns
Status from encrypt/decrypt operation

status_t LTC_DES_EncryptOfb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[8], const uint8_t key[8])

Encrypts DES using OFB block mode.

Encrypts DES using OFB block mode.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes
- iv – Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
- key – Input key to use for encryption

Returns
Status from encrypt/decrypt operation

status_t LTC_DES_DecryptOfb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[8], const uint8_t key[8])

Decrypts DES using OFB block mode.

Decrypts DES using OFB block mode.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes. Must be multiple of 8 bytes.
- iv – Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
- key – Input key to use for decryption

Returns
Status from encrypt/decrypt operation

status_t LTC_DES2_EncryptEcb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t key1[8], const uint8_t key2[8])

Encrypts triple DES using ECB block mode with two keys.

Encrypts triple DES using ECB block mode with two keys.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes. Must be multiple of 8 bytes.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns

Status from encrypt/decrypt operation

status_t LTC_DES2_DecryptEcb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t key1[8], const uint8_t key2[8])

Decrypts triple DES using ECB block mode with two keys.

Decrypts triple DES using ECB block mode with two keys.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes. Must be multiple of 8 bytes.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns

Status from encrypt/decrypt operation

status_t LTC_DES2_EncryptCbc(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const uint8_t key2[8])

Encrypts triple DES using CBC block mode with two keys.

Encrypts triple DES using CBC block mode with two keys.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes
- iv – Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns

Status from encrypt/decrypt operation

status_t LTC_DES2_DecryptCbc(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const uint8_t key2[8])

Decrypts triple DES using CBC block mode with two keys.

Decrypts triple DES using CBC block mode with two keys.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes
- iv – Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns

Status from encrypt/decrypt operation

status_t LTC_DES2_EncryptCfb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const uint8_t key2[8])

Encrypts triple DES using CFB block mode with two keys.

Encrypts triple DES using CFB block mode with two keys.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes
- iv – Input initial block.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns

Status from encrypt/decrypt operation

status_t LTC_DES2_DecryptCfb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const uint8_t key2[8])

Decrypts triple DES using CFB block mode with two keys.

Decrypts triple DES using CFB block mode with two keys.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes
- iv – Input initial block.

- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns
Status from encrypt/decrypt operation

status_t LTC_DES2_EncryptOfb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const uint8_t key2[8])

Encrypts triple DES using OFB block mode with two keys.

Encrypts triple DES using OFB block mode with two keys.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes
- iv – Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns
Status from encrypt/decrypt operation

status_t LTC_DES2_DecryptOfb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const uint8_t key2[8])

Decrypts triple DES using OFB block mode with two keys.

Decrypts triple DES using OFB block mode with two keys.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes
- iv – Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle

Returns
Status from encrypt/decrypt operation

status_t LTC_DES3_EncryptEcb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t key1[8], const uint8_t key2[8], const uint8_t key3[8])

Encrypts triple DES using ECB block mode with three keys.

Encrypts triple DES using ECB block mode with three keys.

Parameters

- base – LTC peripheral base address

- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes. Must be multiple of 8 bytes.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns
Status from encrypt/decrypt operation

status_t LTC_DES3_DecryptEcb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t key1[8], const uint8_t key2[8], const uint8_t key3[8])

Decrypts triple DES using ECB block mode with three keys.

Decrypts triple DES using ECB block mode with three keys.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes. Must be multiple of 8 bytes.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns
Status from encrypt/decrypt operation

status_t LTC_DES3_EncryptCbc(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const uint8_t key2[8], const uint8_t key3[8])

Encrypts triple DES using CBC block mode with three keys.

Encrypts triple DES using CBC block mode with three keys.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input data in bytes
- iv – Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns
Status from encrypt/decrypt operation

status_t LTC_DES3_DecryptCbc(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const uint8_t key2[8], const uint8_t key3[8])

Decrypts triple DES using CBC block mode with three keys.

Decrypts triple DES using CBC block mode with three keys.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes
- iv – Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns

Status from encrypt/decrypt operation

status_t LTC_DES3_EncryptCfb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const uint8_t key2[8], const uint8_t key3[8])

Encrypts triple DES using CFB block mode with three keys.

Encrypts triple DES using CFB block mode with three keys.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and ouput data in bytes
- iv – Input initial block.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns

Status from encrypt/decrypt operation

status_t LTC_DES3_DecryptCfb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const uint8_t key2[8], const uint8_t key3[8])

Decrypts triple DES using CFB block mode with three keys.

Decrypts triple DES using CFB block mode with three keys.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext

- size – Size of input data in bytes
- iv – Input initial block.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns
Status from encrypt/decrypt operation

status_t LTC_DES3_EncryptOfb(LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const uint8_t key2[8], const uint8_t key3[8])

Encrypts triple DES using OFB block mode with three keys.

Encrypts triple DES using OFB block mode with three keys.

Parameters

- base – LTC peripheral base address
- plaintext – Input plaintext to encrypt
- ciphertext – **[out]** Output ciphertext
- size – Size of input and output data in bytes
- iv – Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns
Status from encrypt/decrypt operation

status_t LTC_DES3_DecryptOfb(LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[8], const uint8_t key1[8], const uint8_t key2[8], const uint8_t key3[8])

Decrypts triple DES using OFB block mode with three keys.

Decrypts triple DES using OFB block mode with three keys.

Parameters

- base – LTC peripheral base address
- ciphertext – Input ciphertext to decrypt
- plaintext – **[out]** Output plaintext
- size – Size of input and output data in bytes
- iv – Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
- key1 – First input key for key bundle
- key2 – Second input key for key bundle
- key3 – Third input key for key bundle

Returns
Status from encrypt/decrypt operation

LTC_DES_KEY_SIZE
LTC DES key size - 64 bits.

LTC_DES_IV_SIZE
LTC DES IV size - 8 bytes.

2.50 LTC HASH driver

enum _ltc_hash_algo_t
Supported cryptographic block cipher functions for HASH creation

Values:

enumerator kLTC_Cmac
CMAC (AES engine)

enumerator kLTC_Sha1
SHA_1 (MDHA engine)

enumerator kLTC_Sha224
SHA_224 (MDHA engine)

enumerator kLTC_Sha256
SHA_256 (MDHA engine)

typedef enum *_ltc_hash_algo_t* ltc_hash_algo_t
Supported cryptographic block cipher functions for HASH creation

typedef struct *_ltc_hash_ctx_t* ltc_hash_ctx_t
Storage type used to save hash context.

status_t LTC_HASH_Init(LTC_Type *base, *ltc_hash_ctx_t* *ctx, *ltc_hash_algo_t* algo, const uint8_t *key, uint32_t keySize)
Initialize HASH context.

This function initialize the HASH. Key shall be supplied if the underlaying algoritm is AES XCBC-MAC or CMAC. Key shall be NULL if the underlaying algoritm is SHA.

For XCBC-MAC, the key length must be 16. For CMAC, the key length can be the AES key lengths supported by AES engine. For MDHA the key length argument is ignored.

Parameters

- base – LTC peripheral base address
- ctx – **[out]** Output hash context
- algo – Underlaying algorithm to use for hash computation.
- key – Input key (NULL if underlaying algorithm is SHA)
- keySize – Size of input key in bytes

Returns
Status of initialization

status_t LTC_HASH_Update(*ltc_hash_ctx_t* *ctx, const uint8_t *input, uint32_t inputSize)
Add data to current HASH.

Add data to current HASH. This can be called repeatedly with an arbitrary amount of data to be hashed.

Parameters

- ctx – **[inout]** HASH context
- input – Input data
- inputSize – Size of input data in bytes

Returns
Status of the hash update operation

status_t LTC_HASH_Finish(*ltc_hash_ctx_t* *ctx, uint8_t *output, uint32_t *outputSize)

Finalize hashing.

Outputs the final hash and erases the context.

Parameters

- ctx – **[inout]** Input hash context
- output – **[out]** Output hash data
- outputSize – **[out]** Output parameter storing the size of the output hash in bytes

Returns
Status of the hash finish operation

status_t LTC_HASH(LTC_Type *base, *ltc_hash_algo_t* algo, const uint8_t *input, uint32_t inputSize, const uint8_t *key, uint32_t keySize, uint8_t *output, uint32_t *outputSize)

Create HASH on given data.

Perform the full keyed HASH in one function call.

Parameters

- base – LTC peripheral base address
- algo – Block cipher algorithm to use for CMAC creation
- input – Input data
- inputSize – Size of input data in bytes
- key – Input key
- keySize – Size of input key in bytes
- output – **[out]** Output hash data
- outputSize – **[out]** Output parameter storing the size of the output hash in bytes

Returns
Status of the one call hash operation.

LTC_HASH_CTX_SIZE

LTC HASH Context size.

struct _ltc_hash_ctx_t

#include <fsl_ltc.h> Storage type used to save hash context.

2.51 LTC PKHA driver

enum _ltc_pkha_timing_t

Use of timing equalized version of a PKHA function.

Values:

enumerator kLTC_PKHA_NoTimingEqualized
Normal version of a PKHA operation

enumerator kLTC_PKHA_TimingEqualized
Timing-equalized version of a PKHA operation

enum _ltc_pkha_f2m_t
Integer vs binary polynomial arithmetic selection.

Values:

enumerator kLTC_PKHA_IntegerArith
Use integer arithmetic

enumerator kLTC_PKHA_F2mArith
Use binary polynomial arithmetic

enum _ltc_pkha_montgomery_form_t
Montgomery or normal PKHA input format.

Values:

enumerator kLTC_PKHA_NormalValue
PKHA number is normal integer

enumerator kLTC_PKHA_MontgomeryFormat
PKHA number is in montgomery format

typedef struct *_ltc_pkha_ecc_point_t* ltc_pkha_ecc_point_t
PKHA ECC point structure

typedef enum *_ltc_pkha_timing_t* ltc_pkha_timing_t
Use of timing equalized version of a PKHA function.

typedef enum *_ltc_pkha_f2m_t* ltc_pkha_f2m_t
Integer vs binary polynomial arithmetic selection.

typedef enum *_ltc_pkha_montgomery_form_t* ltc_pkha_montgomery_form_t
Montgomery or normal PKHA input format.

int LTC_PKHA_CompareBigNum(const uint8_t *a, size_t sizeA, const uint8_t *b, size_t sizeB)
Compare two PKHA big numbers.

Compare two PKHA big numbers. Return 1 for a > b, -1 for a < b and 0 if they are same. PKHA big number is lsbyte first. Thus the comparison starts at msbyte which is the last member of tested arrays.

Parameters

- a – First integer represented as an array of bytes, lsbyte first.
- sizeA – Size in bytes of the first integer.
- b – Second integer represented as an array of bytes, lsbyte first.
- sizeB – Size in bytes of the second integer.

Returns
1 if a > b.

Returns
-1 if a < b.

Returns
0 if a = b.

status_t LTC_PKHA_NormalToMontgomery(LTC_Type *base, const uint8_t *N, uint16_t sizeN, uint8_t *A, uint16_t *sizeA, uint8_t *B, uint16_t *sizeB, uint8_t *R2, uint16_t *sizeR2, *ltc_pkha_timing_t* equalTime, *ltc_pkha_f2m_t* arithType)

Converts from integer to Montgomery format.

This function computes R2 mod N and optionally converts A or B into Montgomery format of A or B.

Parameters

- base – LTC peripheral base address
- N – modulus
- sizeN – size of N in bytes
- A – **[inout]** The first input in non-Montgomery format. Output Montgomery format of the first input.
- sizeA – **[inout]** pointer to size variable. On input it holds size of input A in bytes. On output it holds size of Montgomery format of A in bytes.
- B – **[inout]** Second input in non-Montgomery format. Output Montgomery format of the second input.
- sizeB – **[inout]** pointer to size variable. On input it holds size of input B in bytes. On output it holds size of Montgomery format of B in bytes.
- R2 – **[out]** Output Montgomery factor R2 mod N.
- sizeR2 – **[out]** pointer to size variable. On output it holds size of Montgomery factor R2 mod N in bytes.
- equalTime – Run the function time equalized or no timing equalization.
- arithType – Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

status_t LTC_PKHA_MontgomeryToNormal(LTC_Type *base, const uint8_t *N, uint16_t sizeN, uint8_t *A, uint16_t *sizeA, uint8_t *B, uint16_t *sizeB, *ltc_pkha_timing_t* equalTime, *ltc_pkha_f2m_t* arithType)

Converts from Montgomery format to int.

This function converts Montgomery format of A or B into int A or B.

Parameters

- base – LTC peripheral base address
- N – modulus.
- sizeN – size of N modulus in bytes.
- A – **[inout]** Input first number in Montgomery format. Output is non-Montgomery format.
- sizeA – **[inout]** pointer to size variable. On input it holds size of the input A in bytes. On output it holds size of non-Montgomery A in bytes.
- B – **[inout]** Input first number in Montgomery format. Output is non-Montgomery format.
- sizeB – **[inout]** pointer to size variable. On input it holds size of the input B in bytes. On output it holds size of non-Montgomery B in bytes.

- equalTime – Run the function time equalized or no timing equalization.
- arithType – Type of arithmetic to perform (integer or F2m)

Returns
Operation status.

status_t LTC_PKHA_ModAdd(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize, *ltc_pkha_f2m_t* arithType)

Performs modular addition - (A + B) mod N.

This function performs modular addition of (A + B) mod N, with either integer or binary polynomial (F2m) inputs. In the F2m form, this function is equivalent to a bitwise XOR and it is functionally the same as subtraction.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- B – second addend (integer or binary polynomial)
- sizeB – Size of B in bytes
- N – modulus. For F2m operation this can be NULL, as N is ignored during F2m polynomial addition.
- sizeN – Size of N in bytes. This must be given for both integer and F2m polynomial additions.
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- arithType – Type of arithmetic to perform (integer or F2m)

Returns
Operation status.

status_t LTC_PKHA_ModSub1(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize)

Performs modular subtraction - (A - B) mod N.

This function performs modular subtraction of (A - B) mod N with integer inputs.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- B – second addend (integer or binary polynomial)
- sizeB – Size of B in bytes
- N – modulus
- sizeN – Size of N in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes

Returns
Operation status.

status_t LTC_PKHA_ModSub2(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize)

Performs modular subtraction - (B - A) mod N.

This function performs modular subtraction of (B - A) mod N, with integer inputs.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- B – second addend (integer or binary polynomial)
- sizeB – Size of B in bytes
- N – modulus
- sizeN – Size of N in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes

Returns

Operation status.

status_t LTC_PKHA_ModMul(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize, *ltc_pkha_f2m_t* arithType, *ltc_pkha_montgomery_form_t* montIn, *ltc_pkha_montgomery_form_t* montOut, *ltc_pkha_timing_t* equalTime)

Performs modular multiplication - (A x B) mod N.

This function performs modular multiplication with either integer or binary polynomial (F2m) inputs. It can optionally specify whether inputs and/or outputs will be in Montgomery form or not.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- B – second addend (integer or binary polynomial)
- sizeB – Size of B in bytes
- N – modulus.
- sizeN – Size of N in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- arithType – Type of arithmetic to perform (integer or F2m)
- montIn – Format of inputs
- montOut – Format of output
- equalTime – Run the function time equalized or no timing equalization. This argument is ignored for F2m modular multiplication.

Returns
Operation status.

status_t LTC_PKHA_ModExp(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *N, uint16_t sizeN, const uint8_t *E, uint16_t sizeE, uint8_t *result, uint16_t *resultSize, *ltc_pkha_f2m_t* arithType, *ltc_pkha_montgomery_form_t* montIn, *ltc_pkha_timing_t* equalTime)

Performs modular exponentiation - (A^E) mod N.

This function performs modular exponentiation with either integer or binary polynomial (F2m) inputs.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- N – modulus
- sizeN – Size of N in bytes
- E – exponent
- sizeE – Size of E in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- montIn – Format of A input (normal or Montgomery)
- arithType – Type of arithmetic to perform (integer or F2m)
- equalTime – Run the function time equalized or no timing equalization.

Returns
Operation status.

status_t LTC_PKHA_ModRed(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize, *ltc_pkha_f2m_t* arithType)

Performs modular reduction - (A) mod N.

This function performs modular reduction with either integer or binary polynomial (F2m) inputs.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- N – modulus
- sizeN – Size of N in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- arithType – Type of arithmetic to perform (integer or F2m)

Returns
Operation status.

status_t LTC_PKHA_ModInv(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize, *ltc_pkha_f2m_t* arithType)

Performs modular inversion - (A^-1) mod N.

This function performs modular inversion with either integer or binary polynomial (F2m) inputs.

Parameters

- base – LTC peripheral base address
- A – first addend (integer or binary polynomial)
- sizeA – Size of A in bytes
- N – modulus
- sizeN – Size of N in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- arithType – Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

status_t LTC_PKHA_ModR2(LTC_Type *base, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize, *ltc_pkha_f2m_t* arithType)

Computes integer Montgomery factor R^2 mod N.

This function computes a constant to assist in converting operands into the Montgomery residue system representation.

Parameters

- base – LTC peripheral base address
- N – modulus
- sizeN – Size of N in bytes
- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- arithType – Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

status_t LTC_PKHA_GCD(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize, *ltc_pkha_f2m_t* arithType)

Calculates the greatest common divisor - GCD (A, N).

This function calculates the greatest common divisor of two inputs with either integer or binary polynomial (F2m) inputs.

Parameters

- base – LTC peripheral base address
- A – first value (must be smaller than or equal to N)
- sizeA – Size of A in bytes
- N – second value (must be non-zero)
- sizeN – Size of N in bytes

- result – **[out]** Output array to store result of operation
- resultSize – **[out]** Output size of operation in bytes
- arithType – Type of arithmetic to perform (integer or F2m)

Returns
Operation status.

status_t LTC_PKHA_PrimalityTest(LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, bool *res)

Executes Miller-Rabin primality test.

This function calculates whether or not a candidate prime number is likely to be a prime.

Parameters

- base – LTC peripheral base address
- A – initial random seed
- sizeA – Size of A in bytes
- B – number of trial runs
- sizeB – Size of B in bytes
- N – candidate prime integer
- sizeN – Size of N in bytes
- res – **[out]** True if the value is likely prime or false otherwise

Returns
Operation status.

status_t LTC_PKHA_ECC_PointAdd(LTC_Type *base, const *ltc_pkha_ecc_point_t* *A, const *ltc_pkha_ecc_point_t* *B, const uint8_t *N, const uint8_t *R2modN, const uint8_t *aCurveParam, const uint8_t *bCurveParam, uint8_t size, *ltc_pkha_f2m_t* arithType, *ltc_pkha_ecc_point_t* *result)

Adds elliptic curve points - A + B.

This function performs ECC point addition over a prime field (Fp) or binary field (F2m) using affine coordinates.

Parameters

- base – LTC peripheral base address
- A – Left-hand point
- B – Right-hand point
- N – Prime modulus of the field
- R2modN – NULL (the function computes R2modN internally) or pointer to pre-computed R2modN (obtained from LTC_PKHA_ModR2() function).
- aCurveParam – A parameter from curve equation
- bCurveParam – B parameter from curve equation (constant)
- size – Size in bytes of curve points and parameters
- arithType – Type of arithmetic to perform (integer or F2m)
- result – **[out]** Result point

Returns
Operation status.

status_t LTC_PKHA_ECC_PointDouble(LTC_Type *base, const *ltc_pkha_ecc_point_t* *B, const uint8_t *N, const uint8_t *aCurveParam, const uint8_t *bCurveParam, uint8_t size, *ltc_pkha_f2m_t* arithType, *ltc_pkha_ecc_point_t* *result)

Doubles elliptic curve points - B + B.

This function performs ECC point doubling over a prime field (Fp) or binary field (F2m) using affine coordinates.

Parameters

- base – LTC peripheral base address
- B – Point to double
- N – Prime modulus of the field
- aCurveParam – A parameter from curve equation
- bCurveParam – B parameter from curve equation (constant)
- size – Size in bytes of curve points and parameters
- arithType – Type of arithmetic to perform (integer or F2m)
- result – **[out]** Result point

Returns

Operation status.

status_t LTC_PKHA_ECC_PointMul(LTC_Type *base, const *ltc_pkha_ecc_point_t* *A, const uint8_t *E, uint8_t sizeE, const uint8_t *N, const uint8_t *R2modN, const uint8_t *aCurveParam, const uint8_t *bCurveParam, uint8_t size, *ltc_pkha_timing_t* equalTime, *ltc_pkha_f2m_t* arithType, *ltc_pkha_ecc_point_t* *result, bool *infinity)

Multiplies an elliptic curve point by a scalar - E x (A0, A1).

This function performs ECC point multiplication to multiply an ECC point by a scalar integer multiplier over a prime field (Fp) or a binary field (F2m).

Parameters

- base – LTC peripheral base address
- A – Point as multiplicand
- E – Scalar multiple
- sizeE – The size of E, in bytes
- N – Modulus, a prime number for the Fp field or Irreducible polynomial for F2m field.
- R2modN – NULL (the function computes R2modN internally) or pointer to pre-computed R2modN (obtained from LTC_PKHA_ModR2() function).
- aCurveParam – A parameter from curve equation
- bCurveParam – B parameter from curve equation (C parameter for operation over F2m).
- size – Size in bytes of curve points and parameters
- equalTime – Run the function time equalized or no timing equalization.
- arithType – Type of arithmetic to perform (integer or F2m)
- result – **[out]** Result point

- infinity – **[out]** Output true if the result is point of infinity, and false otherwise. Writing of this output will be ignored if the argument is NULL.

Returns
Operation status.

struct __ltc_pkha_ecc_point_t
#include <fsl_ltc.h> PKHA ECC point structure

Public Members

uint8_t *X
X coordinate (affine)

uint8_t *Y
Y coordinate (affine)

2.52 LTC Blocking APIs

2.53 MCM: Miscellaneous Control Module

FSL_MCM_DRIVER_VERSION
MCM driver version.

Enum _mcm_interrupt_flag. Interrupt status flag mask. .

Values:

enumerator kMCM_CacheWriteBuffer
Cache Write Buffer Error Enable.

enumerator kMCM_ParityError
Cache Parity Error Enable.

enumerator kMCM_FPUInvalidOperation
FPU Invalid Operation Interrupt Enable.

enumerator kMCM_FPUDivideByZero
FPU Divide-by-zero Interrupt Enable.

enumerator kMCM_FPUOverflow
FPU Overflow Interrupt Enable.

enumerator kMCM_FPUUnderflow
FPU Underflow Interrupt Enable.

enumerator kMCM_FPUInexact
FPU Inexact Interrupt Enable.

enumerator kMCM_FPUInputDenormalInterrupt
FPU Input Denormal Interrupt Enable.

typedef union *_mcm_buffer_fault_attribute* mcm_buffer_fault_attribute_t
The union of buffer fault attribute.

typedef union *_mcm_lmem_fault_attribute* mcm_lmem_fault_attribute_t
The union of LMEM fault attribute.

static inline void MCM_EnableCrossbarRoundRobin(MCM_Type *base, bool enable)

Enables/Disables crossbar round robin.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable crossbar round robin.
  - **true** Enable crossbar round robin.
  - **false** disable crossbar round robin.

static inline void MCM_EnableInterruptStatus(MCM_Type *base, uint32_t mask)

Enables the interrupt.

Parameters

- base – MCM peripheral base address.
- mask – Interrupt status flags mask(_mcm_interrupt_flag).

static inline void MCM_DisableInterruptStatus(MCM_Type *base, uint32_t mask)

Disables the interrupt.

Parameters

- base – MCM peripheral base address.
- mask – Interrupt status flags mask(_mcm_interrupt_flag).

static inline uint16_t MCM_GetInterruptStatus(MCM_Type *base)

Gets the Interrupt status .

Parameters

- base – MCM peripheral base address.

static inline void MCM_ClearCacheWriteBufferErroStatus(MCM_Type *base)

Clears the Interrupt status .

Parameters

- base – MCM peripheral base address.

static inline uint32_t MCM_GetBufferFaultAddress(MCM_Type *base)

Gets buffer fault address.

Parameters

- base – MCM peripheral base address.

static inline void MCM_GetBufferFaultAttribute(MCM_Type *base, *mcm_buffer_fault_attribute_t* *bufferfault)

Gets buffer fault attributes.

Parameters

- base – MCM peripheral base address.
- bufferfault – Structure to store the result.

static inline uint32_t MCM_GetBufferFaultData(MCM_Type *base)

Gets buffer fault data.

Parameters

- base – MCM peripheral base address.

static inline void MCM_LimitCodeCachePeripheralWriteBuffering(MCM_Type *base, bool enable)

Limit code cache peripheral write buffering.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable limit code cache peripheral write buffering.
  - **true** Enable limit code cache peripheral write buffering.
  - **false** disable limit code cache peripheral write buffering.

static inline void MCM_BypassFixedCodeCacheMap(MCM_Type *base, bool enable)

Bypass fixed code cache map.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable bypass fixed code cache map.
  - **true** Enable bypass fixed code cache map.
  - **false** disable bypass fixed code cache map.

static inline void MCM_EnableCodeBusCache(MCM_Type *base, bool enable)

Enables/Disables code bus cache.

Parameters

- base – MCM peripheral base address.
- enable – Used to disable/enable code bus cache.
  - **true** Enable code bus cache.
  - **false** disable code bus cache.

static inline void MCM_ForceCodeCacheToNoAllocation(MCM_Type *base, bool enable)

Force code cache to no allocation.

Parameters

- base – MCM peripheral base address.
- enable – Used to force code cache to allocation or no allocation.
  - **true** Force code cache to no allocation.
  - **false** Force code cache to allocation.

static inline void MCM_EnableCodeCacheWriteBuffer(MCM_Type *base, bool enable)

Enables/Disables code cache write buffer.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable code cache write buffer.
  - **true** Enable code cache write buffer.
  - **false** Disable code cache write buffer.

static inline void MCM_ClearCodeBusCache(MCM_Type *base)

Clear code bus cache.

Parameters

- base – MCM peripheral base address.

static inline void MCM_EnablePcParityFaultReport(MCM_Type *base, bool enable)

Enables/Disables PC Parity Fault Report.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable PC Parity Fault Report.
  - **true** Enable PC Parity Fault Report.
  - **false** disable PC Parity Fault Report.

static inline void MCM_EnablePcParity(MCM_Type *base, bool enable)

Enables/Disables PC Parity.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable PC Parity.
  - **true** Enable PC Parity.
  - **false** disable PC Parity.

static inline void MCM_LockConfigState(MCM_Type *base)

Lock the configuration state.

Parameters

- base – MCM peripheral base address.

static inline void MCM_EnableCacheParityReporting(MCM_Type *base, bool enable)

Enables/Disables cache parity reporting.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable cache parity reporting.
  - **true** Enable cache parity reporting.
  - **false** disable cache parity reporting.

static inline uint32_t MCM_GetLmemFaultAddress(MCM_Type *base)

Gets LMEM fault address.

Parameters

- base – MCM peripheral base address.

static inline void MCM_GetLmemFaultAttribute(MCM_Type *base, *mcm_lmem_fault_attribute_t* *lmemFault)

Get LMEM fault attributes.

Parameters

- base – MCM peripheral base address.
- lmemFault – Structure to store the result.

static inline uint64_t MCM_GetLmemFaultData(MCM_Type *base)

Gets LMEM fault data.

Parameters

- base – MCM peripheral base address.

MCM_LMFATR_TYPE_MASK

MCM_LMFATR_MODE_MASK

MCM_LMFATR_BUFF_MASK

MCM_LMFATR_CACH_MASK

MCM_ISCR_STAT_MASK

FSL_COMPONENT_ID

union __mcm_buffer_fault_attribute

#include <fsl_mcm.h> The union of buffer fault attribute.

Public Members

uint32_t attribute

Indicates the faulting attributes, when a properly-enabled cache write buffer error interrupt event is detected.

struct *_mcm_buffer_fault_attribute._mcm_buffer_fault_attribut* attribute_memory

struct __mcm_buffer_fault_attribut

#include <fsl_mcm.h>

Public Members

uint32_t busErrorDataAccessType

Indicates the type of cache write buffer access.

uint32_t busErrorPrivilegeLevel

Indicates the privilege level of the cache write buffer access.

uint32_t busErrorSize

Indicates the size of the cache write buffer access.

uint32_t busErrorAccess

Indicates the type of system bus access.

uint32_t busErrorMasterID

Indicates the crossbar switch bus master number of the captured cache write buffer bus error.

uint32_t busErrorOverrun

Indicates if another cache write buffer bus error is detected.

union __mcm_lmem_fault_attribute

#include <fsl_mcm.h> The union of LMEM fault attribute.

Public Members

uint32_t attribute

Indicates the attributes of the LMEM fault detected.

struct *_mcm_lmem_fault_attribute._mcm_lmem_fault_attribut* attribute_memory

struct __mcm_lmem_fault_attribut

#include <fsl_mcm.h>

Public Members

uint32_t parityFaultProtectionSignal
Indicates the features of parity fault protection signal.

uint32_t parityFaultMasterSize
Indicates the parity fault master size.

uint32_t parityFaultWrite
Indicates the parity fault is caused by read or write.

uint32_t backdoorAccess
Indicates the LMEM access fault is initiated by core access or backdoor access.

uint32_t parityFaultSyndrome
Indicates the parity fault syndrome.

uint32_t overrun
Indicates the number of faultss.

2.54 MSCM: Miscellaneous System Control

FSL_MSCM_DRIVER_VERSION
MSCM driver version 2.0.0.

typedef struct *_mscm_uid* mscm_uid_t

static inline void MSCM_GetUID(MSCM_Type *base, *mscm_uid_t* *uid)
Get MSCM UID.

Parameters

- base – MSCM peripheral base address.
- uid – Pointer to an uid struct.

static inline void MSCM_SetSecureIrqParameter(MSCM_Type *base, const uint32_t parameter)
Set MSCM Secure Irq.

Parameters

- base – MSCM peripheral base address.
- parameter – Value to be write to SECURE_IRQ.

static inline uint32_t MSCM_GetSecureIrq(MSCM_Type *base)
Get MSCM Secure Irq.

Parameters

- base – MSCM peripheral base address.

Returns
MSCM Secure Irq.

FSL_COMPONENT_ID

struct _mscm_uid
#include <fsl_mscm.h>

2.55 PORT: Port Control and Interrupts

static inline void PORT_GetVersionInfo(PORT_Type *base, *port_version_info_t* *info)

Get PORT version information.

Parameters

- base – PORT peripheral base pointer
- info – PORT version information

static inline void PORT_SecletPortVoltageRange(PORT_Type *base, *port_voltage_range_t* range)

Get PORT version information.

Note: : PORTA_CONFIG[RANGE] controls the voltage ranges of Port A, B, and C. Read or write PORTB_CONFIG[RANGE] and PORTC_CONFIG[RANGE] does not take effect.

Parameters

- base – PORT peripheral base pointer
- range – port voltage range

static inline void PORT_SetPinConfig(PORT_Type *base, uint32_t pin, const *port_pin_config_t* *config)

Sets the port PCR register.

This is an example to define an input pin or output pin PCR configuration.

```
// Define a digital input pin PCR configuration
port_pin_config_t config = {
    kPORT_PullUp,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainDisable,
    kPORT_LowDriveStrength,
    kPORT_MuxAsGpio,
    kPORT_UnLockRegister,
};
```

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- config – PORT PCR register configuration structure.

static inline void PORT_SetMultiplePinsConfig(PORT_Type *base, uint32_t mask, const *port_pin_config_t* *config)

Sets the port PCR register for multiple pins.

This is an example to define input pins or output pins PCR configuration.

```
Define a digital input pin PCR configuration
port_pin_config_t config = {
    kPORT_PullUp ,
    kPORT_PullEnable,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainDisable,
    kPORT_LowDriveStrength,
```

(continues on next page)

(continued from previous page)

```
    kPORT_MuxAsGpio,
    kPORT_UnlockRegister,
};
```

Parameters

- base – PORT peripheral base pointer.
- mask – PORT pin number macro.
- config – PORT PCR register configuration structure.

static inline void PORT_SetPinMux(PORT_Type *base, uint32_t pin, *port_mux_t* mux)

Configures the pin muxing.

Note: : This function is NOT recommended to use together with the PORT_SetPinsConfig, because the PORT_SetPinsConfig need to configure the pin mux anyway (Otherwise the pin mux is reset to zero : kPORT_PinDisabledOrAnalog). This function is recommended to use to reset the pin mux

---

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- mux – pin muxing slot selection.
  - kPORT_PinDisabledOrAnalog: Pin disabled or work in analog function.
  - kPORT_MuxAsGpio : Set as GPIO.
  - kPORT_MuxAlt2 : chip-specific.
  - kPORT_MuxAlt3 : chip-specific.
  - kPORT_MuxAlt4 : chip-specific.
  - kPORT_MuxAlt5 : chip-specific.
  - kPORT_MuxAlt6 : chip-specific.
  - kPORT_MuxAlt7 : chip-specific.

static inline void PORT_EnablePinsDigitalFilter(PORT_Type *base, uint32_t mask, bool enable)

Enables the digital filter in one port, each bit of the 32-bit register represents one pin.

Parameters

- base – PORT peripheral base pointer.
- mask – PORT pin number macro.
- enable – PORT digital filter configuration.

static inline void PORT_SetDigitalFilterConfig(PORT_Type *base, const *port_digital_filter_config_t* *config)

Sets the digital filter in one port, each bit of the 32-bit register represents one pin.

Parameters

- base – PORT peripheral base pointer.
- config – PORT digital filter configuration structure.

static inline void PORT_SetPinDriveStrength(PORT_Type *base, uint32_t pin, uint8_t strength)

Configures the port pin drive strength.

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- strength – PORT pin drive strength
  - kPORT_LowDriveStrength = 0U - Low-drive strength is configured.
  - kPORT_HighDriveStrength = 1U - High-drive strength is configured.

static inline void PORT_EnablePinDoubleDriveStrength(PORT_Type *base, uint32_t pin, bool enable)

Enables the port pin double drive strength.

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- enable – PORT pin drive strength configuration.

static inline void PORT_SetPinPullValue(PORT_Type *base, uint32_t pin, uint8_t value)

Configures the port pin pull value.

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- value – PORT pin pull value
  - kPORT_LowPullResistor = 0U - Low internal pull resistor value is selected.
  - kPORT_HighPullResistor = 1U - High internal pull resistor value is selected.

static inline uint32_t PORT_GetEFTDetectFlags(PORT_Type *base)

Get EFT detect flags.

Parameters

- base – PORT peripheral base pointer

Returns

EFT detect flags

static inline void PORT_EnableEFTDetectInterrupts(PORT_Type *base, uint32_t interrupt)

Enable EFT detect interrupts.

Parameters

- base – PORT peripheral base pointer
- interrupt – EFT detect interrupt

static inline void PORT_DisableEFTDetectInterrupts(PORT_Type *base, uint32_t interrupt)

Disable EFT detect interrupts.

Parameters

- base – PORT peripheral base pointer
- interrupt – EFT detect interrupt

static inline void PORT_ClearAllLowEFTDetectors(PORT_Type *base)

Clear all low EFT detector.

Note: : Port B and Port C pins share the same EFT detector clear control from PORTC_EDCR register. Any write to the PORTB_EDCR does not take effect.

Parameters

- base – PORT peripheral base pointer

static inline void PORT_ClearAllHighEFTDetectors(PORT_Type *base)

Clear all high EFT detector.

Parameters

- base – PORT peripheral base pointer

FSL_PORT_DRIVER_VERSION

PORT driver version.

enum _port_pull

Internal resistor pull feature selection.

Values:

enumerator kPORT_PullDisable

Internal pull-up/down resistor is disabled.

enumerator kPORT_PullDown

Internal pull-down resistor is enabled.

enumerator kPORT_PullUp

Internal pull-up resistor is enabled.

enum _port_pull_value

Internal resistor pull value selection.

Values:

enumerator kPORT_LowPullResistor

Low internal pull resistor value is selected.

enumerator kPORT_HighPullResistor

High internal pull resistor value is selected.

enum _port_slew_rate

Slew rate selection.

Values:

enumerator kPORT_FastSlewRate

Fast slew rate is configured.

enumerator kPORT_SlowSlewRate

Slow slew rate is configured.

enum _port_open_drain_enable

Open Drain feature enable/disable.

Values:

enumerator kPORT_OpenDrainDisable

Open drain output is disabled.

enumerator kPORT_OpenDrainEnable
Open drain output is enabled.

enum _port_passive_filter_enable
Passive filter feature enable/disable.

Values:

enumerator kPORT_PassiveFilterDisable
Passive input filter is disabled.

enumerator kPORT_PassiveFilterEnable
Passive input filter is enabled.

enum _port_drive_strength
Configures the drive strength.

Values:

enumerator kPORT_LowDriveStrength
Low-drive strength is configured.

enumerator kPORT_HighDriveStrength
High-drive strength is configured.

enum _port_drive_strength1
Configures the drive strength1.

Values:

enumerator kPORT_NormalDriveStrength
Normal drive strength

enumerator kPORT_DoubleDriveStrength
Double drive strength

enum _port_lock_register
Unlock/lock the pin control register field[15:0].

Values:

enumerator kPORT_UnlockRegister
Pin Control Register fields [15:0] are not locked.

enumerator kPORT_LockRegister
Pin Control Register fields [15:0] are locked.

enum _port_mux
Pin mux selection.

Values:

enumerator kPORT_PinDisabledOrAnalog
Corresponding pin is disabled, but is used as an analog pin.

enumerator kPORT_MuxAsGpio
Corresponding pin is configured as GPIO.

enumerator kPORT_MuxAlt0
Chip-specific

enumerator kPORT_MuxAlt1
Chip-specific

enumerator kPORT_MuxAlt2
Chip-specific

enumerator kPORT_MuxAlt3
Chip-specific

enumerator kPORT_MuxAlt4
Chip-specific

enumerator kPORT_MuxAlt5
Chip-specific

enumerator kPORT_MuxAlt6
Chip-specific

enumerator kPORT_MuxAlt7
Chip-specific

enumerator kPORT_MuxAlt8
Chip-specific

enumerator kPORT_MuxAlt9
Chip-specific

enumerator kPORT_MuxAlt10
Chip-specific

enumerator kPORT_MuxAlt11
Chip-specific

enumerator kPORT_MuxAlt12
Chip-specific

enumerator kPORT_MuxAlt13
Chip-specific

enumerator kPORT_MuxAlt14
Chip-specific

enumerator kPORT_MuxAlt15
Chip-specific

enum _port_digital_filter_clock_source
Digital filter clock source selection.

Values:

enumerator kPORT_BusClock
Digital filters are clocked by the bus clock.

enumerator kPORT_LpoClock
Digital filters are clocked by the 1 kHz LPO clock.

enum _port_voltage_range
PORT voltage range.

Values:

enumerator kPORT_VoltageRange1Dot71V_3Dot6V
Port voltage range is 1.71 V - 3.6 V.

enumerator kPORT_VoltageRange2Dot70V_3Dot6V
Port voltage range is 2.70 V - 3.6 V.

typedef enum *_port_mux* port_mux_t
Pin mux selection.

typedef enum *_port_digital_filter_clock_source* port_digital_filter_clock_source_t
Digital filter clock source selection.

typedef struct *_port_digital_filter_config* port_digital_filter_config_t
PORT digital filter feature configuration definition.

typedef struct *_port_pin_config* port_pin_config_t
PORT pin configuration structure.

typedef struct *_port_version_info* port_version_info_t
PORT version information.

typedef enum *_port_voltage_range* port_voltage_range_t
PORT voltage range.

FSL_COMPONENT_ID

struct _port_digital_filter_config
#include <fsl_port.h> PORT digital filter feature configuration definition.

Public Members

uint32_t digitalFilterWidth
Set digital filter width

port_digital_filter_clock_source_t clockSource
Set digital filter clockSource

struct _port_pin_config
#include <fsl_port.h> PORT pin configuration structure.

Public Members

uint16_t pullSelect
No-pull/pull-down/pull-up select

uint16_t pullValueSelect
Pull value select

uint16_t slewRate
Fast/slow slew rate Configure

uint16_t passiveFilterEnable
Passive filter enable/disable

uint16_t openDrainEnable
Open drain enable/disable

uint16_t driveStrength
Fast/slow drive strength configure

uint16_t driveStrength1
Normal/Double drive strength enable/disable

uint16_t lockRegister
Lock/unlock the PCR field[15:0]

struct _port_version_info
#include <fsl_port.h> PORT version information.

Public Members

uint16_t feature
Feature Specification Number.

uint8_t minor
Minor Version Number.

uint8_t major
Major Version Number.

2.56 RTC: Real Time Clock

void RTC_Init(RTC_Type *base, const *rtc_config_t* *config)
Ungates the RTC clock and configures the peripheral for basic operation.

This function issues a software reset if the timer invalid flag is set.

Note: This API should be called at the beginning of the application using the RTC driver.

Parameters

- base – RTC peripheral base address
- config – Pointer to the user's RTC configuration structure.

static inline void RTC_Deinit(RTC_Type *base)
Stops the timer and gate the RTC clock.

Parameters

- base – RTC peripheral base address

void RTC_GetDefaultConfig(*rtc_config_t* *config)
Fills in the RTC config struct with the default settings.

The default values are as follows.

```
config->clockOutput = false;
config->wakeupSelect = false;
config->updateMode = false;
config->supervisorAccess = false;
config->compensationInterval = 0;
config->compensationTime = 0;
```

Parameters

- config – Pointer to the user's RTC configuration structure.

status_t RTC_SetDatetime(RTC_Type *base, const *rtc_datetime_t* *datetime)
Sets the RTC date and time according to the given time structure.

The RTC counter must be stopped prior to calling this function because writes to the RTC seconds register fail if the RTC counter is running.

Parameters

- base – RTC peripheral base address
- datetime – Pointer to the structure where the date and time details are stored.

Returns
kStatus_Success: Success in setting the time and starting the RTC kStatus_InvalidArgument: Error because the datetime format is incorrect

void RTC_GetDatetime(RTC_Type *base, *rtc_datetime_t* *datetime)

Gets the RTC time and stores it in the given time structure.

Parameters

- base – RTC peripheral base address
- datetime – Pointer to the structure where the date and time details are stored.

status_t RTC_SetAlarm(RTC_Type *base, const *rtc_datetime_t* *alarmTime)

Sets the RTC alarm time.

The function checks whether the specified alarm time is greater than the present time. If not, the function does not set the alarm and returns an error.

Parameters

- base – RTC peripheral base address
- alarmTime – Pointer to the structure where the alarm time is stored.

Returns
kStatus_Success: success in setting the RTC alarm kStatus_InvalidArgument: Error because the alarm datetime format is incorrect kStatus_Fail: Error because the alarm time has already passed

void RTC_GetAlarm(RTC_Type *base, *rtc_datetime_t* *datetime)

Returns the RTC alarm time.

Parameters

- base – RTC peripheral base address
- datetime – Pointer to the structure where the alarm date and time details are stored.

void RTC_EnableInterrupts(RTC_Type *base, uint32_t mask)

Enables the selected RTC interrupts.

Parameters

- base – RTC peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration rtc_interrupt_enable_t

void RTC_DisableInterrupts(RTC_Type *base, uint32_t mask)

Disables the selected RTC interrupts.

Parameters

- base – RTC peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration rtc_interrupt_enable_t

uint32_t RTC_GetEnabledInterrupts(RTC_Type *base)

Gets the enabled RTC interrupts.

Parameters

- base – RTC peripheral base address

Returns
The enabled interrupts. This is the logical OR of members of the enumeration rtc_interrupt_enable_t

uint32_t RTC_GetStatusFlags(RTC_Type *base)

Gets the RTC status flags.

Parameters

- base – RTC peripheral base address

Returns
The status flags. This is the logical OR of members of the enumeration rtc_status_flags_t

void RTC_ClearStatusFlags(RTC_Type *base, uint32_t mask)

Clears the RTC status flags.

Parameters

- base – RTC peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration rtc_status_flags_t

static inline void RTC_EnableLPOClock(RTC_Type *base, bool enable)

Enable/Disable RTC 1kHz LPO clock.

Note: After setting this bit, RTC prescaler increments using the LPO 1kHz clock and not the RTC 32kHz crystal clock.

Parameters

- base – RTC peripheral base address
- enable – Enable/Disable RTC 1kHz LPO clock

static inline void RTC_StartTimer(RTC_Type *base)

Starts the RTC time counter.

After calling this function, the timer counter increments once a second provided SR[TOF] or SR[TIF] are not set.

Parameters

- base – RTC peripheral base address

static inline void RTC_StopTimer(RTC_Type *base)

Stops the RTC time counter.

RTC's seconds register can be written to only when the timer is stopped.

Parameters

- base – RTC peripheral base address

void RTC_GetMonotonicCounter(RTC_Type *base, uint64_t *counter)

Reads the values of the Monotonic Counter High and Monotonic Counter Low and returns them as a single value.

Parameters

- base – RTC peripheral base address
- counter – Pointer to variable where the value is stored.

void RTC_SetMonotonicCounter(RTC_Type *base, uint64_t counter)

Writes values Monotonic Counter High and Monotonic Counter Low by decomposing the given single value. The Monotonic Overflow Flag in RTC_SR is cleared due to the API.

Parameters

- base – RTC peripheral base address
- counter – Counter value

status_t RTC_IncrementMonotonicCounter(RTC_Type *base)

Increments the Monotonic Counter by one.

Increments the Monotonic Counter (registers RTC_MCLR and RTC_MCHR accordingly) by setting the monotonic counter enable (MER[MCE]) and then writing to the RTC_MCLR register. A write to the monotonic counter low that causes it to overflow also increments the monotonic counter high.

Parameters

- base – RTC peripheral base address

Returns

kStatus_Success: success kStatus_Fail: error occurred, either time invalid or monotonic overflow flag was found

FSL_RTC_DRIVER_VERSION

Version 2.4.0

enum _rtc_interrupt_enable

List of RTC interrupts.

Values:

enumerator kRTC_TimeInvalidInterruptEnable

Time invalid interrupt.

enumerator kRTC_TimeOverflowInterruptEnable

Time overflow interrupt.

enumerator kRTC_AlarmInterruptEnable

Alarm interrupt.

enumerator kRTC_MonotonicOverflowInterruptEnable

Monotonic Overflow Interrupt Enable

enumerator kRTC_SecondsInterruptEnable

Seconds interrupt.

enumerator kRTC_TestModeInterruptEnable

enumerator kRTC_FlashSecurityInterruptEnable

enumerator kRTC_TamperPinInterruptEnable

enumerator kRTC_SecurityModuleInterruptEnable

enumerator kRTC_LossOfClockInterruptEnable

enum _rtc_status_flags

List of RTC flags.

Values:

enumerator kRTC_TimeInvalidFlag

Time invalid flag

enumerator kRTC_TimeOverflowFlag
Time overflow flag

enumerator kRTC_AlarmFlag
Alarm flag

enumerator kRTC_MonotonicOverflowFlag
Monotonic Overflow Flag

enumerator kRTC_TamperInterruptDetectFlag
Tamper interrupt detect flag

enumerator kRTC_TestModeFlag

enumerator kRTC_FlashSecurityFlag

enumerator kRTC_TamperPinFlag

enumerator kRTC_SecurityTamperFlag

enumerator kRTC_LossOfClockTamperFlag

enum _rtc_osc_cap_load
List of RTC Oscillator capacitor load settings.

Values:

enumerator kRTC_Capacitor_2p
2 pF capacitor load

enumerator kRTC_Capacitor_4p
4 pF capacitor load

enumerator kRTC_Capacitor_8p
8 pF capacitor load

enumerator kRTC_Capacitor_16p
16 pF capacitor load

enum _rtc_timer_seconds_interrupt_frequency
List of RTC Timer Seconds Interrupt Frequencies.

Values:

enumerator kRTC_TimerSecondsFrequency1Hz
Timer seconds frequency is 1Hz

enumerator kRTC_TimerSecondsFrequency2Hz
Timer seconds frequency is 2Hz

enumerator kRTC_TimerSecondsFrequency4Hz
Timer seconds frequency is 4Hz

enumerator kRTC_TimerSecondsFrequency8Hz
Timer seconds frequency is 8Hz

enumerator kRTC_TimerSecondsFrequency16Hz
Timer seconds frequency is 16Hz

enumerator kRTC_TimerSecondsFrequency32Hz
Timer seconds frequency is 32Hz

enumerator kRTC_TimerSecondsFrequency64Hz
Timer seconds frequency is 64Hz

enumerator kRTC_TimerSecondsFrequency128Hz
Timer seconds frequency is 128Hz

typedef enum *_rtc_interrupt_enable* rtc_interrupt_enable_t
List of RTC interrupts.

typedef enum *_rtc_status_flags* rtc_status_flags_t
List of RTC flags.

typedef enum *_rtc_osc_cap_load* rtc_osc_cap_load_t
List of RTC Oscillator capacitor load settings.

typedef enum *_rtc_timer_seconds_interrupt_frequency* rtc_timer_seconds_interrupt_frequency_t
List of RTC Timer Seconds Interrupt Frequencies.

typedef struct *_rtc_datetime* rtc_datetime_t
Structure is used to hold the date and time.

typedef struct *_rtc_pin_config* rtc_pin_config_t
RTC pin config structure.

typedef struct *_rtc_config* rtc_config_t
RTC config structure.

This structure holds the configuration settings for the RTC peripheral. To initialize this structure to reasonable defaults, call the RTC_GetDefaultConfig() function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

static inline uint32_t RTC_GetTamperTimeSeconds(RTC_Type *base)
Get the RTC tamper time seconds.

Parameters

- base – RTC peripheral base address

static inline void RTC_SetOscCapLoad(RTC_Type *base, uint32_t capLoad)
This function sets the specified capacitor configuration for the RTC oscillator.

Parameters

- base – RTC peripheral base address
- capLoad – Oscillator loads to enable. This is a logical OR of members of the enumeration rtc_osc_cap_load_t

static inline void RTC_Reset(RTC_Type *base)
Performs a software reset on the RTC module.

This resets all RTC registers except for the SWR bit and the RTC_WAR and RTC_RAR registers. The SWR bit is cleared by software explicitly clearing it.

Parameters

- base – RTC peripheral base address

static inline void RTC_EnableWakeUpPin(RTC_Type *base, bool enable)
Enables or disables the RTC Wakeup Pin Operation.

This function enable or disable RTC Wakeup Pin. The wakeup pin is optional and not available on all devices.

Parameters

- base – RTC_Type base pointer.
- enable – true to enable, false to disable.

static inline void RTC_EnableClockOutput(RTC_Type *base, bool enable)

Enables or disables the RTC 32 kHz clock output.

This function enables or disables the RTC 32 kHz clock output.

Parameters

- base – RTC_Type base pointer.
- enable – true to enable, false to disable.

void RTC_SetTimerSecondsInterruptFrequency(RTC_Type *base, *rtc_timer_seconds_interrupt_frequency_t* freq)

Sets the RTC timer seconds interrupt frequency.

This function sets the RTC timer seconds interrupt frequency.

Parameters

- base – RTC peripheral base address
- freq – The timer seconds interrupt frequency. This is a member of the enumeration rtc_timer_seconds_interrupt_frequency_t

struct __rtc__datetime

#include <fsl_rtc.h> Structure is used to hold the date and time.

Public Members

uint16_t year

Range from 1970 to 2099.

uint8_t month

Range from 1 to 12.

uint8_t day

Range from 1 to 31 (depending on month).

uint8_t hour

Range from 0 to 23.

uint8_t minute

Range from 0 to 59.

uint8_t second

Range from 0 to 59.

struct __rtc__pin__config

#include <fsl_rtc.h> RTC pin config structure.

Public Members

bool inputLogic

true: Tamper pin input data is logic one. false: Tamper pin input data is logic zero.

bool pinActiveLow

true: Tamper pin is active low. false: Tamper pin is active high.

bool filterEnable

true: Input filter is enabled on the tamper pin. false: Input filter is disabled on the tamper pin.

bool pullSelectNegate
true: Tamper pin pull resistor direction will negate the tamper pin. false: Tamper pin pull resistor direction will assert the tamper pin.

bool pullEnable
true: Pull resistor is enabled on tamper pin. false: Pull resistor is disabled on tamper pin.

struct _rtc_config
#include <fsl_rtc.h> RTC config structure.

This structure holds the configuration settings for the RTC peripheral. To initialize this structure to reasonable defaults, call the RTC_GetDefaultConfig() function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

Public Members

bool clockOutput
true: The 32 kHz clock is not output to other peripherals; false: The 32 kHz clock is output to other peripherals

bool wakeupSelect
true: Wakeup pin outputs the 32 KHz clock; false:Wakeup pin used to wakeup the chip

bool updateMode
true: Registers can be written even when locked under certain conditions, false: No writes allowed when registers are locked

bool supervisorAccess
true: Non-supervisor accesses are allowed; false: Non-supervisor accesses are not supported

uint32_t compensationInterval
Compensation interval that is written to the CIR field in RTC TCR Register

uint32_t compensationTime
Compensation time that is written to the TCR field in RTC TCR Register

2.57 SEMA42: Hardware Semaphores Driver

FSL_SEMA42_DRIVER_VERSION
SEMA42 driver version.

SEMA42 status return codes.

Values:

enumerator kStatus_SEMA42_Busy
SEMA42 gate has been locked by other processor.

enumerator kStatus_SEMA42_Reseting
SEMA42 gate reseting is ongoing.

enum _sema42_gate_status
SEMA42 gate lock status.

Values:

enumerator kSEMA42_Unlocked
The gate is unlocked.

enumerator kSEMA42_LockedByProc0
The gate is locked by processor 0.

enumerator kSEMA42_LockedByProc1
The gate is locked by processor 1.

enumerator kSEMA42_LockedByProc2
The gate is locked by processor 2.

enumerator kSEMA42_LockedByProc3
The gate is locked by processor 3.

enumerator kSEMA42_LockedByProc4
The gate is locked by processor 4.

enumerator kSEMA42_LockedByProc5
The gate is locked by processor 5.

enumerator kSEMA42_LockedByProc6
The gate is locked by processor 6.

enumerator kSEMA42_LockedByProc7
The gate is locked by processor 7.

enumerator kSEMA42_LockedByProc8
The gate is locked by processor 8.

enumerator kSEMA42_LockedByProc9
The gate is locked by processor 9.

enumerator kSEMA42_LockedByProc10
The gate is locked by processor 10.

enumerator kSEMA42_LockedByProc11
The gate is locked by processor 11.

enumerator kSEMA42_LockedByProc12
The gate is locked by processor 12.

enumerator kSEMA42_LockedByProc13
The gate is locked by processor 13.

enumerator kSEMA42_LockedByProc14
The gate is locked by processor 14.

typedef enum *_sema42_gate_status* sema42_gate_status_t
SEMA42 gate lock status.

void SEMA42_Init(SEMA42_Type *base)
Initializes the SEMA42 module.

This function initializes the SEMA42 module. It only enables the clock but does not reset the gates because the module might be used by other processors at the same time. To reset the gates, call either SEMA42_ResetGate or SEMA42_ResetAllGates function.

Parameters

- base – SEMA42 peripheral base address.

void SEMA42_Deinit(SEMA42_Type *base)

De-initializes the SEMA42 module.

This function de-initializes the SEMA42 module. It only disables the clock.

Parameters

- base – SEMA42 peripheral base address.

status_t SEMA42_TryLock(SEMA42_Type *base, uint8_t gateNum, uint8_t procNum)

Tries to lock the SEMA42 gate.

This function tries to lock the specific SEMA42 gate. If the gate has been locked by another processor, this function returns an error code.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number to lock.
- procNum – Current processor number.

Return values

- kStatus_Success – Lock the sema42 gate successfully.
- kStatus_SEMA42_Busy – Sema42 gate has been locked by another processor.

status_t SEMA42_Lock(SEMA42_Type *base, uint8_t gateNum, uint8_t procNum)

Locks the SEMA42 gate.

This function locks the specific SEMA42 gate. If the gate has been locked by other processors, this function waits until it is unlocked and then lock it.

If SEMA42_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return kStatus_Timeout.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number to lock.
- procNum – Current processor number.

Return values

- kStatus_Success – The gate was successfully locked.
- kStatus_Timeout – Timeout occurred while waiting for the gate to be unlocked.

Returns

status_t

static inline void SEMA42_Unlock(SEMA42_Type *base, uint8_t gateNum)

Unlocks the SEMA42 gate.

This function unlocks the specific SEMA42 gate. It only writes unlock value to the SEMA42 gate register. However, it does not check whether the SEMA42 gate is locked by the current processor or not. As a result, if the SEMA42 gate is not locked by the current processor, this function has no effect.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number to unlock.

static inline *sema42_gate_status_t* SEMA42_GetGateStatus(SEMA42_Type *base, uint8_t gateNum)

Gets the status of the SEMA42 gate.

This function checks the lock status of a specific SEMA42 gate.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number.

Returns

status Current status.

status_t SEMA42_ResetGate(SEMA42_Type *base, uint8_t gateNum)

Resets the SEMA42 gate to an unlocked status.

This function resets a SEMA42 gate to an unlocked status.

Parameters

- base – SEMA42 peripheral base address.
- gateNum – Gate number.

Return values

- kStatus_Success – SEMA42 gate is reset successfully.
- kStatus_SEMA42_Reseting – Some other reset process is ongoing.

static inline *status_t* SEMA42_ResetAllGates(SEMA42_Type *base)

Resets all SEMA42 gates to an unlocked status.

This function resets all SEMA42 gate to an unlocked status.

Parameters

- base – SEMA42 peripheral base address.

Return values

- kStatus_Success – SEMA42 is reset successfully.
- kStatus_SEMA42_Reseting – Some other reset process is ongoing.

SEMA42_GATE_NUM_RESET_ALL

The number to reset all SEMA42 gates.

SEMA42_GATEn(base, n)

SEMA42 gate n register address.

The SEMA42 gates are sorted in the order 3, 2, 1, 0, 7, 6, 5, 4, ... not in the order 0, 1, 2, 3, 4, 5, 6, 7, ... The macro SEMA42_GATEn gets the SEMA42 gate based on the gate index.

The input gate index is XOR'ed with 3U: 0 ^ 3 = 3 1 ^ 3 = 2 2 ^ 3 = 1 3 ^ 3 = 0 4 ^ 3 = 7 5 ^ 3 = 6 6 ^ 3 = 5 7 ^ 3 = 4 ...

SEMA42_BUSY_POLL_COUNT

Maximum polling iterations for SEMA42 waiting loops.

This parameter defines the maximum number of iterations for any polling loop in the SEMA42 driver code before timing out and returning an error.

It applies to all waiting loops in SEMA42 driver, such as waiting for a gate to be unlocked, waiting for a reset to complete, or waiting for a resource to become available.

This is a count of loop iterations, not a time-based value.

If defined as 0, polling loops will continue indefinitely until their exit condition is met, which could potentially cause the system to hang if hardware doesn't respond or if a resource is never released.

2.58 SFA: Signal Frequency Analyser

void SFA_GetDefaultConfig(*sfa_config_t* *config)

Fill the SFA configuration structure with default settings.

The default values are:

```
config->mode = kSFA_FrequencyMeasurement0;
config->cutSelect = kSFA_CUTSelect0;
config->refSelect = kSFA_REFSelect0;
config->prediv = 0U;
config->trigStart = kSFA_TriggerStartSelect0;
config->startPolarity = kSFA_TriggerStartPolarityRiseEdge;
config->trigEnd = kSFA_TriggerEndSelect0;
config->endPolarity = kSFA_TriggerEndPolarityRiseEdge;
config->enableTrigMeasurement = false;
config->enableCUTPin = false;
config->cutTarget = 0xffffU;
config->refTarget = 0xffffffffU;
```

Parameters

- config – Pointer to the user configuration structure.

void SFA_Init(SFA_Type *base)

Initialize SFA.

Parameters

- base – SFA peripheral base address.

status_t SFA_Deinit(SFA_Type *base)

Clear counter, disable SFA and gate the SFA clock.

Parameters

- base – SFA peripheral base address.

Return values

- kStatus_Success – run success.
- kStatus_Timeout – timeout occurs.

static inline void SFA_EnableCUTPin(SFA_Type *base, bool enable)

Control the connection of the clock under test to an external pin.

Parameters

- base – SFA peripheral base address.
- enable – true: connect the clock under test and external pin. false: Disconnect the clock under test and external pin.

static inline uint32_t SFA_GetStatusFlags(SFA_Type *base)

Get SFA status flags.

Parameters

- base – SFA peripheral base address.

void SFA_ClearStatusFlag(SFA_Type *base, uint32_t mask)

Clear the SFA status flags.

Note: To clear kSFA_RefStoppedFlag, kSFA_CUTStoppedFlag, kSFA_MeasurementStartedFlag, and kSFA_ReferenceCounterTimeOutFlag, each counter will also be cleared.

Parameters

- base – SFA peripheral base address.
- mask – SFA status flag mask (see _sfa_status_flags for bit definition).

static inline void SFA_EnableInterrupts(SFA_Type *base, uint32_t mask)

Enable the selected SFA interrupt.

Parameters

- base – SFA peripheral base address.
- mask – The interrupt to enable (see _sfa_interrupts_enable for definition).

static inline void SFA_DisableInterrupts(SFA_Type *base, uint32_t mask)

Disable the selected SFA interrupt.

Parameters

- base – SFA peripheral base address.
- mask – The interrupt to disable (see _sfa_interrupts_enable for definition).

static inline uint8_t SFA_GetMode(SFA_Type *base)

Get SFA measurement mode.

Parameters

- base – SFA peripheral base address.

static inline uint8_t SFA_GetCUTPredivide(SFA_Type *base)

Get CUT predivide value.

Parameters

- base – SFA peripheral base address.

void SFA_InstallCallback(SFA_Type *base, *sfa_callback_t* function)

Install the callback function to be called when IRQ happens or measurement completes.

Parameters

- base – SFA peripheral base address.
- function – the SFA measure completed callback function.

void SFA_SetMeasureConfig(SFA_Type *base, const *sfa_config_t* *config)

Set Measurement options with the passed in configuration structure.

Parameters

- base – SFA peripheral base address.
- config – SFA configuration structure.

status_t SFA_MeasureBlocking(SFA_Type *base)

Start SFA measurement in blocking mode.

Parameters

- base – SFA peripheral base address.

Return values

- kStatus_SFA_MeasurementCompleted – SFA measure completes.
- kStatus_SFA_ReferenceCounterTimeout – reference counter timeout error happens.
- kStatus_SFA_CUTCounterTimeout – CUT counter time out happens.

void SFA_MeasureNonBlocking(SFA_Type *base)

Start measure sequence in NonBlocking mode.

This function performs nonblocking measurement by enabling sfa interrupt (Please enable the FreqGreaterThanMax and FreqLessThanMin interrupts individually as needed). The callback function must be installed before invoking this function.

Note: This function has different functions for different instances.

Parameters

- base – SFA peripheral base address.

void SFA_AbortMeasureSequence(SFA_Type *base)

Abort SFA measurement sequence.

Parameters

- base – SFA peripheral base address.

uint32_t SFA_CalculateFrequencyOrPeriod(SFA_Type *base, uint32_t refFrequency)

Calculate the frequency or period.

Parameters

- base – SFA peripheral base address.
- refFrequency – The reference clock frequency(BUS clock recommended).

static inline uint32_t SFA_GetCUTCounter(SFA_Type *base)

Get current count of the clock under test.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetCUTTargetCount(SFA_Type *base, uint32_t count)

Set the target count for the clock under test.

Parameters

- base – SFA peripheral base address.
- count – target count for CUT.

static inline uint32_t SFA_GetCUTTargetCount(SFA_Type *base)

Get the target count of the clock under test.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetCUTLowLimitClockCount(SFA_Type *base, uint32_t count)

Set CUT low limit clock count.

Parameters

- base – SFA peripheral base address.
- count – low limit count for CUT clock.

static inline uint32_t SFA_GetCUTLowLimitClockCount(SFA_Type *base)

Get CUT low limit clock count.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetCUTHighLimitClockCount(SFA_Type *base, uint32_t count)

Set CUT high limit clock count.

Parameters

- base – SFA peripheral base address.
- count – high limit count for CUT clock.

static inline uint32_t SFA_GetCUTHighLimitClockCount(SFA_Type *base)

Get CUT high limit clock count.

Parameters

- base – SFA peripheral base address.

static inline uint32_t SFA_GetREFCounter(SFA_Type *base)

Get current count of the reference clock.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetREFTargetCount(SFA_Type *base, uint32_t count)

Set the target count for the reference clock.

Parameters

- base – SFA peripheral base address.
- count – target count for reference clock.

static inline uint32_t SFA_GetREFTargetCount(SFA_Type *base)

Get the target count of the reference clock.

Parameters

- base – SFA peripheral base address.

static inline uint32_t SFA_GetREFStartCount(SFA_Type *base)

Get saved reference clock counter which is loaded when measurement start.

Parameters

- base – SFA peripheral base address.

static inline uint32_t SFA_GetREFEndCount(SFA_Type *base)

Get saved reference clock counter which is loaded when measurement complete.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetREFLowLimitClockCount(SFA_Type *base, uint32_t count)

Set REF low limit clock count.

Parameters

- base – SFA peripheral base address.

- count – low limit count for REF clock.

static inline uint32_t SFA_GetREFLowLimitClockCount(SFA_Type *base)

Get REF low limit clock count.

Parameters

- base – SFA peripheral base address.

static inline void SFA_SetREFHighLimitClockCount(SFA_Type *base, uint32_t count)

Set REF high limit clock count.

Parameters

- base – SFA peripheral base address.
- count – high limit count for REF clock.

static inline uint32_t SFA_GetREFHighLimitClockCount(SFA_Type *base)

Get REF high limit clock count.

Parameters

- base – SFA peripheral base address.

FSL_SFA_DRIVER_VERSION

SFA driver version 2.1.4.

SFA_MEASUREMENT_START_TIMEOUT

Max loops to wait for SFA measurement started.

This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

SFA_CUT_COUNTER_STOP_TIMEOUT

Max loops to wait for SFA CUT counter has stopped.

This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

SFA_REF_COUNTER_STOP_TIMEOUT

Max loops to wait for SFA REF counter has stopped.

This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

SFA status return codes.

enumeration _sfa_status

Values:

enumerator kStatus_SFA_MeasurementCompleted

Measurement completed

enumerator kStatus_SFA_ReferenceCounterTimeout

Reference counter timeout

enumerator kStatus_SFA_CUTCounterTimeout

CUT counter timeout

enumerator kStatus_SFA_CUTClockFreqLessThanMinLimit

CUT clock frequency less than minimum limit

enumerator kStatus_SFA_CUTClockFreqGreaterThanMaxLimit

CUT clock frequency greater than maximum limit

enum _sfa_status_flags

List of SFA status flags.

The following status register flags can be cleared on any write to REF_CNT.

- kSFA_RefStoppedFlag
- kSFA_CutStoppedFlag
- kSFA_MeasurementStartedFlag
- kSFA_ReferenceCounterTimeOutFlag

Note: These enums are meant to be OR'd together to from a bit mask.

Values:

enumerator kSFA_RefStoppedFlag
Reference counter stopped flag

enumerator kSFA_CutStoppedFlag
CUT counter stopped flag

enumerator kSFA_MeasurementStartedFlag
Measurement Started flag

enumerator kSFA_ReferenceCounterTimeOutFlag
Reference counter time out flag

enumerator kSFA_InterruptRequestFlag
SFA interrupt request flag

enumerator kSFA_FreqGreaterThanMaxInterruptFlag
FREQ_GT_MAX interrupt flag

enumerator kSFA_FreqLessThanMinInterruptFlag
FREQ_LT_MIN interrupt flag

enumerator kSFA_AllStatusFlags

enum _sfa_interrupts_enable

List of SFA interrupt.

Values:

enumerator kSFA_InterruptEnable
SFA interrupt enable

enumerator kSFA_FreqGreaterThanMaxInterruptEnable
FREQ_GT_MAX interrupt enable

enumerator kSFA_FreqLessThanMinInterruptEnable
FREQ_LT_MIN interrupt enable

enum _sfa_measurement_mode

List of SFA measurement mode(Please check the mode configuration according to the manual).

Values:

enumerator kSFA_FrequencyMeasurement0
Frequency measurement performed with REF frequency > CUT frequency

enumerator kSFA_FrequencyMeasurement1
Frequency measurement performed with REF frequency < CUT frequency

enumerator kSFA_CUTPeriodMeasurement
CUT period measurement performed

enumerator kSFA_TriggerBasedMeasurement
Trigger based measurement performed

enum _sfa_cut_select
List of CUT which is connected to the CUT counter (Please refer to the manual for configuration).

Values:

enumerator kSFA_CUTSelect0

enumerator kSFA_CUTSelect1

enumerator kSFA_CUTSelect2

enumerator kSFA_CUTSelect3

enumerator kSFA_CUTSelect4

enumerator kSFA_CUTSelect5

enumerator kSFA_CUTSelect6

enumerator kSFA_CUTSelect7

enumerator kSFA_CUTSelect8

enumerator kSFA_CUTSelect9

enumerator kSFA_CUTSelect10

enumerator kSFA_CUTSelect11

enumerator kSFA_CUTSelect12

enumerator kSFA_CUTSelect13

enumerator kSFA_CUTSelect14

enumerator kSFA_CUTSelect15

enum _sfa_ref_select
List of REF which is connected to the REF counter (Please refer to the manual for configuration).

Values:

enumerator kSFA_REFSelect0

enumerator kSFA_REFSelect1

enumerator kSFA_REFSelect2

enum _sfa_trigger_start_select
List of Signal MUX for Trigger Based Measurement Start.

Values:

enumerator kSFA_TriggerStartSelect0

enumerator kSFA_TriggerStartSelect1

enum _sfa_trigger_end_select

List of Signal MUX for Trigger Based Measurement End.

Values:

enumerator kSFA_TriggerEndSelect0

enumerator kSFA_TriggerEndSelect1

enum _sfa_trigger_start_polarity

List of Trigger Start Polarity.

Values:

enumerator kSFA_TriggerStartPolarityRiseEdge

Rising edge will begin the measurement sequence

enumerator kSFA_TriggerStartPolarityFallEdge

Falling edge will begin the measurement sequence

enum _sfa_trigger_end_polarity

List of Trigger End Polarity.

Values:

enumerator kSFA_TriggerEndPolarityRiseEdge

Rising edge will end the measurement sequence

enumerator kSFA_TriggerEndPolarityFallEdge

Falling edge will end the measurement sequence

typedef void (*sfa_callback_t)(*status_t* status)

sfa measure completion callback function pointer type

This callback can be used in non blocking IRQHandler. Specify the callback you want in the call to SFA_InstallCallback().

Param status

The runtime measurement status. kStatus_SFA_MeasurementCompleted: The measurement completes. kStatus_SFA_ReferenceCounterTimeout: Reference counter timeout happenes. kStatus_SFA_CUTCounterTimeout: CUT counter timeout happenes.

typedef enum *_sfa_measurement_mode* sfa_measurement_mode_t

List of SFA measurement mode(Please check the mode configuration according to the manual).

typedef enum *_sfa_cut_select* sfa_cut_select_t

List of CUT which is connected to the CUT counter (Please refer to the manual for configuration).

typedef enum *_sfa_ref_select* sfa_ref_select_t

List of REF which is connected to the REF counter (Please refer to the manual for configuration).

typedef enum *_sfa_trigger_start_select* sfa_trigger_start_select_t

List of Signal MUX for Trigger Based Measurement Start.

typedef enum *_sfa_trigger_end_select* sfa_trigger_end_select_t

List of Signal MUX for Trigger Based Measurement End.

typedef enum *_sfa_trigger_start_polarity* sfa_trigger_start_polarity_t

List of Trigger Start Polarity.

typedef enum *_sfa_trigger_end_polarity* sfa_trigger_end_polarity_t

List of Trigger End Polarity.

typedef struct *_sfa_init_config* sfa_config_t

Structure with setting to initialize the SFA module.

This structure holds configuration setting for the SFA peripheral. To initialize this structure to reasonable defaults, call the SFA_GetDefaultConfig() function and pass a pointer to your configuration structure instance.

SFA_CUT_CLK_Enable(val)

struct _sfa_init_config

#include <fsl_sfa.h> Structure with setting to initialize the SFA module.

This structure holds configuration setting for the SFA peripheral. To initialize this structure to reasonable defaults, call the SFA_GetDefaultConfig() function and pass a pointer to your configuration structure instance.

Public Members

sfa_measurement_mode_t mode

measurement mode

sfa_cut_select_t cutSelect

Select clock connected to the clock under test counter

sfa_ref_select_t refSelect

Selcet REF connected the bus clock

uint8_t prediv

Integer divide of the Input CUT signal

sfa_trigger_start_select_t trigStart

Select the signal will be used to end a trigger based measurement

sfa_trigger_start_polarity_t startPolarity

Select the polarity of the start trigger signal

sfa_trigger_end_select_t trigEnd

Select the signal will be used to commence a trigger based measurement

sfa_trigger_end_polarity_t endPolarity

Select the polarity of the end trigger signal

bool enableTrigMeasurement

false: The measurement will start by default with a dummy write to the CUT counter; true : The measurement will start after receiving a dummy write to the REF_CNT followed by receiving the trigger edge

bool enableCUTPin

Control the connection of the clock under test to an external pin.

uint32_t refTarget

Reference counter target counts

uint32_t cutTarget

CUT counter target counts

2.59 SMSCM: Secure Miscellaneous System Control Module

FSL_MSCM_DRIVER_VERSION

SMSCM driver version 2.0.0.

enum _smscm_debug

SMSCM debug enable type.

Values:

enumerator kSMSCM_InvasiveDebug

enumerator kSMSCM_SecureInvasiveDebug

enumerator kSMSCM_NonInvasiveDebug

enumerator kSMSCM_SecureNonInvasiveDebug

enumerator kSMSCM_AltInvasiveDebug

enumerator kSMSCM_AltDebug

enum _smscm_mem

SMSCM On-Chip Memory Descriptor Register.

Values:

enumerator kSMSCM_Mem0

enumerator kSMSCM_Mem2

enumerator kSMSCM_Mem3

enumerator kSMSCM_Mem5

enum _smscm_ecc_ctrl

SMSCM ECC control type of On-Chip Memory.

Values:

enumerator kSMSCM_EccDisable

enumerator kSMSCM_EccEnableOnWrite

enumerator kSMSCM_EccEnableOnRead

enumerator kSMSCM_EccEnableOnWriteAndRead

typedef enum *_smscm_mem* smscm_mem_t

SMSCM On-Chip Memory Descriptor Register.

typedef enum *_smscm_ecc_ctrl* smscm_ecc_ctrl_t

SMSCM ECC control type of On-Chip Memory.

typedef struct *smscm_ecc_fault_attr* smscm_ecc_fault_attr_t

SMSCM attribute.

static inline void SMSCM_EnableDebug(SMSCM_Type *base, uint32_t debugToEnable)

Enable the Debug function. DebugToEnable could be bitwise OR of _smscm_debug.

Parameters

- base – SMSCM peripheral address.
- debugToEnable – debug enable type.

static inline void SMSCM_DisableDebug(SMSCM_Type *base, uint32_t debugToDisable)

Disables the Debug function. DebugToDisable could be bitwise OR of _smscm_debug.

Parameters

- base – SMSCM peripheral address.
- debugToDisable – debug disable type.

static inline void SMSCM_DebugLock(SMSCM_Type *base)

Lock the debug function.

Parameters

- base – SMSCM peripheral base address.

static inline uint32_t SMSCM_GetSecurityCount(SMSCM_Type *base)

Get value in Security Counter Register (SCTR).

Parameters

- base – SMSCM peripheral base address.

Returns

SMSCM SCTR value.

static inline void SMSCM_SetSecurityCount(SMSCM_Type *base, uint32_t val)

Set value in Security Counter Register (SCTR).

Parameters

- base – SMSCM peripheral base address.
- val – SCTR value to set.

static inline void SMSCM_IncreaseSecurityCount(SMSCM_Type *base, uint32_t val)

Write value to be plused in Security Counter Register (SCTR).

The entire contents of the write data word are added to the security counter, and next-state SCTR =current-state SCTR + DATA32.

Parameters

- base – SMSCM peripheral base address.
- val – SCTR value to plus.

static inline void SMSCM_DecreaseSecurityCount(SMSCM_Type *base, uint32_t val)

Write value to be minused in Security Counter Register (SCTR).

The entire contents of the write data word are added to the security counter, and next-state SCTR =current-state SCTR - DATA32.

Parameters

- base – SMSCM peripheral base address.
- val – SCTR value to be minused.

static inline void SMSCM_IncreaseSecurityCountBy1(SMSCM_Type *base)

Increase security counter rigister by 1.

Parameters

- base – SMSCM peripheral base address.

static inline void SMSCM_DecreaseSecurityCountBy1(SMSCM_Type *base)

Decrease security counter rigister by 1.

Parameters

- base – SMSCM peripheral base address.

static inline void SMSCM_LockMemControlReg(SMSCM_Type *base, *smscm_mem_t* mem)

Lock the on-chip memory descriptor. This register bit provides a mechanism to "lock" the configuration state defined by OCMDRn[11:0]. Once asserted, attempted writes to the OCMDRn[11:0] register are ignored until the next reset clears the flag.

Parameters

- base – SMSCM peripheral address.
- mem – Select OCMDRn to enable read-only mode.

static inline void SMSCM_EnableFlashCache(SMSCM_Type *base, bool enable)

Enable or disable the on-chip memory flash cache.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

static inline void SMSCM_EnableFlashInstructionCache(SMSCM_Type *base, bool enable)

Enable flash instruction cache.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

static inline void SMSCM_EnableFlashDataCache(SMSCM_Type *base, bool enable)

Enable flash data cache.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

static inline void SMSCM_ClearFlashCache(SMSCM_Type *base)

Clear the on-chip memory flash cache.

Parameters

- base – SMSCM peripheral address.

static inline void SMSCM_LockFlashIFR1(SMSCM_Type *base)

Lock IFR1 by flash controller.

Parameters

- base – SMSCM peripheral address.

static inline void SMSCM_EnableFlashSpeculate(SMSCM_Type *base, bool enable)

SMSCM Flash Speculate enable.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

static inline void SMSCM_EnableDataPrefetch(SMSCM_Type *base, bool enable)

SMSCM Data Prefetch enable.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

static inline void SMSCM_EnableFlashDataNonCorrectableBusError(SMSCM_Type *base, bool enable)

Disable non-correctable bus errors on flash data fetches.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

static inline void SMSCM_EnableFlashInstructionNonCorrectableBusError(SMSCM_Type *base, bool enable)

Disable non-correctable bus errors on flash instruction fetches.

Parameters

- base – SMSCM peripheral address.
- enable – Use true to enable, false to disable.

static inline void SMSCM_SetMemEccControl(SMSCM_Type *base, *smscm_mem_t* mem, *smscm_ecc_ctrl_t* eccCtrl)

Select ecc control type in OCMDRn.

Parameters

- base – SMSCM peripheral address.
- mem – Select OCMDRn.
- eccCtrl – Select ecc control type.

static inline void SMSCM_EnableEccReport(SMSCM_Type *base)

Enable RAM ECC 1 bit and non-correctable reporting.

Parameters

- base – SMSCM peripheral base address.

static inline bool SMSCM_GetEccValid(SMSCM_Type *base)

Get the ECC location valid states.

Parameters

- base – SMSCM peripheral base address.

Returns

State of ECC Error Location field.

static inline uint8_t SMSCM_GetEccLocation(SMSCM_Type *base)

Get the ECC location.

Parameters

- base – SMSCM peripheral base address.

Returns

ECC fault location.

void SMSCM_ClearEccError(SMSCM_Type *base, uint8_t errLocation)

Clear each 1-bit correctable or non-correctable error.

Parameters

- base – SMSCM peripheral address.
- errLocation – ECC Error Location

static inline uint32_t SMSCM_GetEccAddress(SMSCM_Type *base)

Get the ECC fault address.

Parameters

- base – SMSCM peripheral base address.

Returns

ECC fault address.

void SMSCM_GetEccAttribute(SMSCM_Type *base, *smscm_ecc_fault_attr_t* *eccAttribute)

Get ECC attribute.

Parameters

- base – SMSCM peripheral address.
- eccAttribute – Ecc attribute.

static inline uint32_t SMSCM_GetEccFaultDataHigh(SMSCM_Type *base)

Get ECC Fault Data High.

This read-only field specifies the upper 32-bit read data word (data[63:32]) from the last captured ECCevent. For ECC events that occur in 32-bit RAMs, this 32-bit field will return 32'h0.

Parameters

- base – SMSCM peripheral base address.

Returns

The higher 32-bit read data word.

static inline uint32_t SMSCM_GetEccFaultDataLow(SMSCM_Type *base)

Get ECC Fault Data Low.

Parameters

- base – SMSCM peripheral base address.

Returns

The lower 32-bit read data word.

struct smscm_ecc_fault_attr

#include <fsl_smscm.h> SMSCM attribute.

2.60 SPC: System Power Control driver

SPC status enumeration.

Values:

enumerator kStatus_SPC_Busy

The SPC instance is busy executing any type of power mode transition.

enumerator kStatus_SPC_DCDCLowDriveStrengthIgnore

DCDC Low drive strength setting be ignored for LVD/HVD enabled.

enumerator kStatus_SPC_DCDCPulseRefreshModeIgnore

DCDC Pulse Refresh Mode drive strength setting be ignored for LVD/HVD enabled.

enumerator kStatus_SPC_SYSLDOOverDriveVoltageFail

SYS LDO regulate to Over drive voltage failed for SYS LDO HVD must be disabled.

enumerator kStatus_SPC_SYSLDOLowDriveStrengthIgnore
SYS LDO Low driver strength setting be ignored for LDO LVD/HVD enabled.

enumerator kStatus_SPC_CORELDOLowDriveStrengthIgnore
CORE LDO Low driver strength setting be ignored for LDO LVD/HVD enabled.

enumerator kStatus_SPC_CORELDOVoltageWrong
Core LDO voltage is wrong.

enumerator kStatus_SPC_CORELDOVoltageSetFail
Core LDO voltage set fail.

enumerator kStatus_SPC_BandgapModeWrong
Selected Bandgap Mode wrong.

enum _spc_voltage_detect_flags
Voltage Detect Status Flags.

Values:

enumerator kSPC_IOVDDHighVoltageDetectFlag
IO VDD High-Voltage detect flag.

enumerator kSPC_SystemVDDHighVoltageDetectFlag
System VDD High-Voltage detect flag.

enumerator kSPC_CoreVDDHighVoltageDetectFlag
Core VDD High-Voltage detect flag.

enumerator kSPC_IOVDDLowVoltageDetectFlag
IO VDD Low-Voltage detect flag.

enumerator kSPC_SystemVDDLowVoltageDetectFlag
System VDD Low-Voltage detect flag.

enumerator kSPC_CoreVDDLowVoltageDetectFlag
Core VDD Low-Voltage detect flag.

enum _spc_power_domains
SPC power domain isolation status.

Values:

enumerator kSPC_MAINPowerDomainRetain
Peripherals and IO pads retain in MAIN Power Domain.

enumerator kSPC_WAKEPowerDomainRetain
Peripherals and IO pads retain in WAKE Power Domain.

enumerator kSPC_2P4GPowerDoaminRetain
Peripherals and IO pads retion in 2.4G Power Domain.

enum _spc_power_domain_id
The enumeration of spc power domain, the connected power domain is chip specfic, please refer to chip's RM for details.

Values:

enumerator kSPC_PowerDomain0
Power domain0, the connected power domain is chip specific.

enumerator kSPC_PowerDomain1
Power domain1, the connected power domain is chip specific.

enumerator kSPC_PowerDomain2
Power domain2, the connected power domain is chip specific.

enum _spc_power_domain_low_power_mode
The enumeration of Power domain's low power mode.

Values:

enumerator kSPC_SleepWithSYSClockRunning
Power domain request SLEEP mode with SYS clock running.

enumerator kSPC_SleepWithSysClockOff
Power domain request SLEEP mode with SYS clock off.

enumerator kSPC_DeepSleepSysClockOff
Power domain request DEEP SLEEP mode with SYS clock off.

enumerator kSPC_PowerDownWithSysClockOff
Power domain request POWER DOWN mode with SYS clock off.

enumerator kSPC_DeepPowerDownWithSysClockOff
Power domain request DEEP POWER DOWN mode with SYS clock off.

enum _spc_lowPower_request_pin_polarity
SPC low power request output pin polarity.

Values:

enumerator kSPC_HighTruePolarity
Control the High Polarity of the Low Power Reqest Pin.

enumerator kSPC_LowTruePolarity
Control the Low Polarity of the Low Power Reqest Pin.

enum _spc_lowPower_request_output_override
SPC low power request output override.

Values:

enumerator kSPC_LowPowerRequestNotForced
Not Forced.

enumerator kSPC_LowPowerRequestReserved
Reserved.

enumerator kSPC_LowPowerRequestForcedLow
Forced Low (Ignore LowPower request output polarity setting.)

enumerator kSPC_LowPowerRequestForcedHigh
Forced High (Ignore LowPower request output polarity setting.)

enum _spc_bandgap_mode
SPC Bandgap mode enumeration in Active mode or Low Power mode.

Values:

enumerator kSPC_BandgapDisabled
Bandgap disabled.

enumerator kSPC_BandgapEnabledBufferDisabled
Bandgap enabled with Buffer disabled.

enumerator kSPC_BandgapEnabledBufferEnabled
Bandgap enabled with Buffer enabled.

enumerator kSPC_BandgapReserved
Reserved.

enum _spc_dcdc_voltage_level
DCDC regulator voltage level enumeration in Active mode or Low Power Mode.

Values:

enumerator kSPC_DCDC_SafeModeVoltage
DCDC VDD Regulator regulate to Safe-Mode Voltage.

enumerator kSPC_DCDC_NormalVoltage
DCDC VDD Regulator regulate to Normal Voltage.

enumerator kSPC_DCDC_MidVoltage
DCDC VDD Regulator regulate to Mid Voltage.

enumerator kSPC_DCDC_LowUnderVoltage
DCDC VDD Regulator regulate to Low Under Voltage.

enum _spc_dcdc_drive_strength
DCDC regulator Drive Strength enumeration in Active mode or Low Power Mode.

Values:

enumerator kSPC_DCDC_PulseRefreshMode
DCDC VDD Regulator Drive Strength set to Pulse Refresh Mode. This enum member is only useful for Low Power Mode config.

enumerator kSPC_DCDC_LowDriveStrength
DCDC VDD regulator Drive Strength set to low.

enumerator kSPC_DCDC_NormalDriveStrength
DCDC VDD regulator Drive Strength set to Normal.

enumerator kSPC_DCDC_Reserved
Reserved.

enum _spc_sys_ldo_voltage_level
SYS LDO regulator voltage level enumeration in Active mode.

Values:

enumerator kSPC_SysLDO_NormalVoltage
SYS LDO VDD Regulator regulate to Normal Voltage(1.8V).

enumerator kSPC_SysLDO_OverDriveVoltage
SYS LDO VDD Regulator regulate to Over Drive Voltage(2.5V).

enum _spc_sys_ldo_drive_strength
SYS LDO regulator Drive Strength enumeration in Active mode or Low Power mode.

Values:

enumerator kSPC_SysLDO_LowDriveStrength
SYS LDO VDD regulator Drive Strength set to low.

enumerator kSPC_SysLDO_NormalDriveStrength
SYS LDO VDD regulator Drive Strength set to Normal.

enum _spc_core_ldo_voltage_level
Core LDO regulator voltage level enumeration in Active mode or Low Power mode.

Values:

enumerator kSPC_CoreLDO_NormalVoltage
Core LDO VDD regulator regulate to Normal Voltage.

enumerator kSPC_CoreLDO_MidDriveVoltage
Core LDO VDD regulator regulate to Mid Drive Voltage.

enumerator kSPC_CoreLDO_UnderDriveVoltage
Core LDO VDD regulator regulate to Under Drive Voltage.

enumerator kSPC_CoreLDO_SafeModeVoltage
Core LDO VDD regulator regulate to Safe-Mode Voltage.

enum _spc_core_ldo_drive_strength
CORE LDO VDD regulator Drive Strength enumeration in Low Power mode.

Values:

enumerator kSPC_CoreLDO_LowDriveStrength
Core LDO VDD regulator Drive Strength set to low.

enumerator kSPC_CoreLDO_NormalDriveStrength
Core LDO VDD regulator Drive Strength set to Normal.

enum _spc_low_voltage_level_select
System/IO VDD Low-Voltage Level Select.

Values:

enumerator kSPC_LowVoltageNormalLevel
Trip point set to Normal level.

enumerator kSPC_LowVoltageSafeLevel
Trip point set to Safe level.

enum _spc_sram_operat_voltage
SRAM operate voltage enumeration.

Values:

enumerator kSPC_SRAM_OperatVoltage1P0V
SRAM operate voltage set to 1.0V.

enumerator kSPC_SRAM_OperatVoltage1P1V
SRAM operate voltage set to 1.1V.

enum _spc_hp_request_override_option
The enumeration of high power request override option.

Values:

enumerator kSPC_HpRequestOverrideDisable
Disable high power request override feature.

enumerator kSPC_HpRequestOverride0
Enable high power request override feature and force value as 0.

enumerator kSPC_HpReqestOverride1
Enable high power request override feature and force value as 1.

enum _spc_vdd_core_glitch_ripple_counter_select
Used to select output of 4-bit ripple counter is used to monitor a glitch on VDD core.

Values:

enumerator kSPC_selectBit0Of4bitRippleCounter
Select bit-0 of 4-bit Ripple Counter to detect glitch on VDD Core.

enumerator kSPC_selectBit1Of4bitRippleCounter
Select bit-1 of 4-bit Ripple Counter to detect glitch on VDD Core.

enumerator kSPC_selectBit2Of4bitRippleCounter
Select bit-2 of 4-bit Ripple Counter to detect glitch on VDD Core.

enumerator kSPC_selectBit3Of4bitRippleCounter
Select bit-3 of 4-bit Ripple Counter to detect glitch on VDD Core.

typedef enum *_spc_power_domain_id* spc_power_domain_id_t
The enumeration of spc power domain, the connected power domain is chip specfic, please refer to chip's RM for details.

typedef enum *_spc_power_domain_low_power_mode* spc_power_domain_low_power_mode_t
The enumeration of Power domain's low power mode.

typedef enum *_spc_lowPower_request_pin_polarity* spc_lowpower_request_pin_polarity_t
SPC low power request output pin polarity.

typedef enum *_spc_lowPower_request_output_override* spc_lowpower_request_output_override_t
SPC low power request output override.

typedef enum *_spc_bandgap_mode* spc_bandgap_mode_t
SPC Bandgap mode enumeration in Active mode or Low Power mode.

typedef enum *_spc_dcdc_voltage_level* spc_dcdc_voltage_level_t
DCDC regulator voltage level enumeration in Active mode or Low Power Mode.

typedef enum *_spc_dcdc_drive_strength* spc_dcdc_drive_strength_t
DCDC regulator Drive Strength enumeration in Active mode or Low Power Mode.

typedef enum *_spc_sys_ldo_voltage_level* spc_sys_ldo_voltage_level_t
SYS LDO regulator voltage level enumeration in Active mode.

typedef enum *_spc_sys_ldo_drive_strength* spc_sys_ldo_drive_strength_t
SYS LDO regulator Drive Strength enumeration in Active mode or Low Power mode.

typedef enum *_spc_core_ldo_voltage_level* spc_core_ldo_voltage_level_t
Core LDO regulator voltage level enumeration in Active mode or Low Power mode.

typedef enum *_spc_core_ldo_drive_strength* spc_core_ldo_drive_strength_t
CORE LDO VDD regulator Drive Strength enumeration in Low Power mode.

typedef enum *_spc_low_voltage_level_select* spc_low_voltage_level_select_t
System/IO VDD Low-Voltage Level Select.

typedef enum *_spc_sram_operat_voltage* spc_sram_operat_voltage_t
SRAM operate voltage enumeration.

typedef struct *_spc_lowpower_request_config* spc_lowpower_request_config_t
Low Power Request output pin configuration.

typedef struct *_spc_intergrated_power_switch_config* spc_intergrated_power_switch_config_t
Integrated power switch configuration.

Note: Legacy structure, will be removed.

typedef struct *_spc_active_mode_core_ldo_option* spc_active_mode_core_ldo_option_t
Core LDO regulator options in Active mode.

typedef struct *_spc_active_mode_sys_ldo_option* spc_active_mode_sys_ldo_option_t
System LDO regulator options in Active mode.

typedef struct *_spc_active_mode_dcdc_option* spc_active_mode_dcdc_option_t
DCDC regulator options in Active mode.

typedef struct *_spc_lowpower_mode_core_ldo_option* spc_lowpower_mode_core_ldo_option_t
Core LDO regulator options in Low Power mode.

typedef struct *_spc_lowpower_mode_sys_ldo_option* spc_lowpower_mode_sys_ldo_option_t
System LDO regulator options in Low Power mode.

typedef struct *_spc_lowpower_mode_dcdc_option* spc_lowpower_mode_dcdc_option_t
DCDC regulator options in Low Power mode.

typedef struct *_spc_voltage_detect_option* spc_voltage_detect_option_t
CORE/SYS/IO VDD Voltage Detect options.

typedef struct *_spc_dcdc_burst_config* spc_dcdc_burst_config_t
DCDC Burst configuration.

typedef struct *_spc_core_voltage_detect_config* spc_core_voltage_detect_config_t
Core Voltage Detect configuration.

typedef struct *_spc_system_voltage_detect_config* spc_system_voltage_detect_config_t
System Voltage Detect Configuration.

typedef struct *_spc_io_voltage_detect_config* spc_io_voltage_detect_config_t
IO Voltage Detect Configuration.

typedef struct *_spc_active_mode_regulators_config* spc_active_mode_regulators_config_t
Active mode configuration.

typedef struct *_spc_lowpower_mode_regulators_config* spc_lowpower_mode_regulators_config_t
Low Power Mode configuration.

typedef enum *_spc_hp_request_override_option* spc_hp_override_request_option_t
The enumeration of high power request override option.

typedef *spc_active_mode_dcdc_option_t* spc_hp_mode_dcdc_option_t

typedef *spc_active_mode_sys_ldo_option_t* spc_hp_mode_sys_ldo_option_t

typedef *spc_active_mode_core_ldo_option_t* spc_hp_mode_core_ldo_option_t

typedef *spc_active_mode_regulators_config_t* spc_hp_mode_regulators_config_t

typedef enum *_spc_vdd_core_glitch_ripple_counter_select*
spc_vdd_core_glitch_ripple_counter_select_t
Used to select output of 4-bit ripple counter is used to monitor a glitch on VDD core.

typedef struct *_spc_vdd_core_glitch_detector_config* spc_vdd_core_glitch_detector_config_t
The configuration of VDD Core glitch detector.

SPC_BUSY_TIMEOUT
Max loops to wait for SPC to stop being busy.

The BUSY bitfield will be set when the SPC performs any kind of power mode transition in active mode or any chip low power mode. You need to wait until this flag is cleared before changing the power mode configuration registers. This parameter defines how many loops to check completion before return timeout. If defined as 0, the driver will wait until completion.

SPC_SRAM_ACK_TIMEOUT

Max loops to wait for SPC to stop being busy.

When changing SRAM voltage, need to wait for SRAM voltage update request acknowledgment. This parameter defines how many loops to check completion before return timeout. If defined as 0, the driver will wait until completion.

SPC_DCDC_ACK_TIMEOUT

Max loops to wait for DCDC burst completed.

When the DCDC burst is requested, it is necessary to wait for the DCDC burst to complete. This parameter defines how many loops to check completion before return timeout. If defined as 0, the driver will wait until it completes.

SPC_HP_CNFG_CTRL_OVERRIDE_OPT_MASK

SPC_HP_CNFG_CTRL_OVERRIDE_OPT_SHIFT

SPC_HP_CNFG_CTRL_OVERRIDE_OPT(x)

uint8_t SPC_GetPeriphIOIsolationStatus(SPC_Type *base)

Gets Isolation status for each power domains.

This function gets the status which indicates whether certain peripheral and the IO pads are in a latched state as a result of having been in POWERDOWN mode.

Parameters

- base – SPC peripheral base address.

Returns

Current isolation status for each power domains. See _spc_power_domains for details.

static inline void SPC_ClearPeriphIOIsolationFlag(SPC_Type *base)

Clears peripherals and I/O pads isolation flags for each power domains.

This function clears peripherals and I/O pads isolation flags for each power domains. After recovering from the POWERDOWN mode, user must invoke this function to release the I/O pads and certain peripherals to their normal run mode state. Before invoking this function, user must restore chip configuration in particular pin configuration for enabled WUU wakeup pins.

Parameters

- base – SPC peripheral base address.

static inline bool SPC_GetBusyStatusFlag(SPC_Type *base)

Gets SPC busy status flag.

This function gets SPC busy status flag. When SPC executing any type of power mode transition in ACTIVE mode or any of the SOC low power mode, the SPC busy status flag is set and this function returns true. When changing CORE LDO voltage level and DCDC voltage level in ACTIVE mode, the SPC busy status flag is set and this function return true.

Parameters

- base – SPC peripheral base address.

Returns

Ack busy flag. true - SPC is busy. false - SPC is not busy.

static inline bool SPC_CheckLowPowerReqest(SPC_Type *base)

Checks system low power request.

Note: Only when all power domains request low power mode entry, the result of this function is true. That means when all power domains request low power mode entry, the SPC regulators will be controlled by LP_CFG register.

Parameters

- base – SPC peripheral base address.

Returns

The system low power request check result.

- **true** All power domains have requested low power mode and SPC has entered a low power state and power mode configuration are based on the LP_CFG configuration register.
- **false** SPC in active mode and ACTIVE_CFG register control system power supply.

static inline void SPC_ClearLowPowerRequest(SPC_Type *base)

Clears system low power request, set SPC in active mode.

Parameters

- base – SPC peripheral base address.

static inline bool SPC_CheckPowerSwitchState(SPC_Type *base)

Checks power switch state.

Parameters

- base – SPC peripheral base address.

Returns

The state(ON/OFF) of power switch.

- **true** Indicates the power switch is ON.
- **false** Indicates the power switch is OFF.

spc_power_domain_low_power_mode_t SPC_GetPowerDomainLowPowerMode(SPC_Type *base, *spc_power_domain_id_t* powerDomainId)

Gets selected power domain's requested low power mode.

Parameters

- base – SPC peripheral base address.
- powerDomainId – Power Domain Id, please refer to spc_power_domain_id_t.

Returns

The selected power domain's requested low power mode, please refer to spc_power_domain_low_power_mode_t.

static inline bool SPC_CheckPowerDomainLowPowerRequest(SPC_Type *base, *spc_power_domain_id_t* powerDomainId)

Checks power domain's low power request.

Parameters

- base – SPC peripheral base address.
- powerDomainId – Power Domain Id, please refer to spc_power_domain_id_t.

Returns
The result of power domain's low power request.

- **true** The selected power domain requests low power mode entry.
- **false** The selected power domain does not request low power mode entry.

static inline void SPC_ClearPowerDomainLowPowerRequestFlag(SPC_Type *base, *spc_power_domain_id_t* powerDomainId)

Clears selected power domain's low power request flag.

Parameters

- base – SPC peripheral base address.
- powerDomainId – Power Domain Id, please refer to spc_power_domain_id_t.

void SPC_SetLowPowerRequestConfig(SPC_Type *base, const *spc_lowpower_request_config_t* *config)

Configs Low power request output pin.

This function config the low power request output pin

Parameters

- base – SPC peripheral base address.
- config – Pointer the spc_lowpower_request_config_t structure.

static inline void SPC_SoftwareGatePowerSwitch(SPC_Type *base, bool gate)

Gates/Un-gates power switch in software mode.

Parameters

- base – SPC peripheral base address.
- gate – Used to gate/ungate power switch
  - **true** The power switch will be gated.
  - **false** The power switch will be un-gated.

static inline void SPC_PowerModeControlPowerSwitch(SPC_Type *base)

Gates power switch at low power modes entry, and un-gates power switch at low power mode wakeup.

Parameters

- base – SPC peripheral base address.

static inline void SPC_SetWakeUpValue(SPC_Type *base, uint32_t data)

Set the address of the function/image to be executed if chip wake from power down or deep power down mode.

Note: Data written by this function is used by BootROM to quickly retore ARM Core context, or to switch execution to a defined address in Flash/SRAM on WakeUp.

Note: The first word must be SP, and the second word must be PC.

Note: Please remember to calculate the CRC value of the first 48 bytes of image/function and save the result to REGFILE1->REG[0]. The BootROM will check this CRC value, if authenticated successfully then the image/function will be executed.

Parameters

- base – SPC peripheral base address.
- data – The address of the function/image to be executed if wakeup from low power mode.

static inline uint32_t SPC_GetWakeUpValue(SPC_Type *base)

Gets back the WakeUp value.

Parameters

- base – SPC peripheral base address.

Returns

The WakeUp value.

static inline bool SPC_CheckHPCfgSelected(SPC_Type *base)

Check if HP_CFG selected as active configuration register.

Parameters

- base – SPC peripheral base address.

Return values

- false – ACTIVE_CFG selected as the active configuration register.
- true – HP_CFG selected as the active configuration register.

static inline void SPC_EnableHighPowerRequest(SPC_Type *base, bool enable)

Enable/disable high power request feature.

Parameters

- base – SPC peripheral base address.
- enable – Used to specify enable/disable the high power request feature:
  - **true** Enable high power request;
  - **false** Disable high power reques.

static inline void SPC_OverrideHighPowerRequest(SPC_Type *base, *spc_hp_override_request_option_t* opt)

Override high power request manually.

Parameters

- base – SPC peripheral base address.
- opt – Specify the option of high power override request, please refer to spc_hp_override_request_option_t.

static inline *spc_core_ldo_voltage_level_t* SPC_GetHighPowerModeCoreLDOVDDVoltageLevel(SPC_Type *base)

Get voltage level of CORE LDO in high power mode.

Parameters

- base – SPC peripheral base address.

Returns
The voltage level of CORE LDO in high power mode, please refer to spc_core_ldo_voltage_level_t.

static inline *spc_bandgap_mode_t* SPC_GetHighPowerModeBandgapMode(SPC_Type *base)

Get bandgap mode in high power mode.

Parameters

- base – SPC peripheral base address.

Returns
The bandgap mode in high power mode, please refer to spc_bandgap_mode_t.

static inline uint32_t SPC_GetHighPowerModeVoltageDetectStatus(SPC_Type *base)

Get enabled state of all voltage detectors.

Parameters

- base – SPC peripheral base address.

Returns
All enabled status of all voltage detectors, 1b1 means the corrsponding voltage detector is enabled.

status_t SPC_SetHighPowerModeBandgapModeConfig(SPC_Type *base, *spc_bandgap_mode_t* mode)

Set bandgap mode in high power mode.

Parameters

- base – SPC peripheral base address.
- mode – Specify the bandgap mode in high power mode.

Return values

- kStatus_SPC_BandgapModeWrong – The Bandgap can not be disabled in high power mode.
- kStatus_Success – Config Bandgap mode in high power mode successful.

static inline void SPC_EnableHighPowerModeCMPBandgapBuffer(SPC_Type *base, bool enable)

Enable/disable CMP buffer in high power mode.

Parameters

- base – SPC peripheral base address.
- enable – Used to enable/disable CMP buffer:
  - **true** Enable CMP buffer in high power mode;
  - **false** Disable CMP buffer in high power mode.

static inline void SPC_DisableHighPowerModeVddCoreGlitchDetect(SPC_Type *base, bool disable)

Disable/enable VDD Core Glitch detect feature in high power mode.

Parameters

- base – SPC peripheral base address.
- disable – Used to disable/enable VDD Core Glitch detect feature:
  - **true** Disable VDD Core Glitch detect feature;
  - **false** Enable VDD Core Glitch detect feature.

status_t SPC_SetHighPowerModeCoreLDORegulatorConfig(SPC_Type *base, const *spc_hp_mode_core_ldo_option_t* *option)

Configure CORE LDO regulator in high power mode.

Parameters

- base – SPC peripheral base address.
- option – Pointer to the CORE LDO regulator configuration, please refer to spc_hp_mode_core_ldo_option_t.

Return values

- kStatus_Success – Config Core LDO regulator in High power mode successful.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_CORELDOLowDriveStrengthIgnore – If any voltage detect enabled, core_ldo's drive strength can not set to low.
- kStatus_SPC_CORELDOVoltageWrong – The selected voltage level in high power mode is not allowed.
- kStatus_Timeout – Timeout occurs while waiting completion.

status_t SPC_SetHighPowerModeSystemLDORegulatorConfig(SPC_Type *base, const *spc_hp_mode_sys_ldo_option_t* *option)

Configure System LDO regulator in high power mode.

Parameters

- base – SPC peripheral base address.
- option – Pointer to the SYSTEM LDO regulator configuration, please refer to spc_hp_mode_sys_ldo_option_t.

Return values

- kStatus_Success – Config System LDO regulator in Active power mode successful.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_SYSLDOOverDriveVoltageFail – Fail to regulator to Over Drive Voltage.
- kStatus_SPC_SYSLDOLowDriveStrengthIgnore – Set driver strength to Low will be ignored.
- kStatus_Timeout – Timeout occurs while waiting completion.

status_t SPC_SetHighPowerModeDCDCRegulatorConfig(SPC_Type *base, const *spc_hp_mode_dcdc_option_t* *option)

Configure DCDC regulator in high power mode.

Parameters

- base – SPC peripheral base address.
- option – Pointer to the DCDC regulator configuration, please refer to spc_hp_mode_dcdc_option_t.

Return values

- kStatus_Success – Config DCDC regulator in Active power mode successful.

- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_DCDCLowDriveStrengthIgnore – Set driver strength to Low will be ignored.
- kStatus_Timeout – Timeout occurs while waiting completion.

status_t SPC_SetHighPowerModeRegulatorsConfig(SPC_Type *base, const *spc_hp_mode_regulators_config_t* *config)

Set configuration of regulators in high power mode.

Parameters

- base – SPC peripheral base address.
- config – Pointer to the regulator configuration, please refer to spc_hp_mode_regulators_config_t.

Return values

- kStatus_Success – Config regulators in High power mode successful.
- kStatus_SPC_BandgapModeWrong – The bandgap mode setting in high power mode is wrong.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_CORELDOVoltageWrong – The selected voltage level in high mode is not allowed.
- kStatus_SPC_SYSLDOOverDriveVoltageFail – Fail to regulator to Over Drive Voltage.
- kStatus_SPC_SYSLDOLowDriveStrengthIgnore – Set driver strength to Low will be ignored.
- kStatus_SPC_DCDCLowDriveStrengthIgnore – Set driver strength to Low will be ignored.

static inline void SPC_EnableHighPowerModeCoreLowVoltageDetect(SPC_Type *base, bool enable)

Enable/disable low voltage detect for VDD_CORE in high power mode.

Parameters

- base – SPC peripheral base address.
- enable – Used to enable/disable low voltage detect feature for VDD_CORE in high power mode:
  - **true** Enable low voltage detect feature for VDD_CORE in high power mode;
  - **false** Disable low voltage detect feature for VDD_CORE in high power mode.

static inline void SPC_EnableHighPowerModeCoreHighVoltageDetect(SPC_Type *base, bool enable)

Enable/disable high voltage detect for VDD_CORE in high power mode.

Parameters

- base – SPC peripheral base address.
- enable – Used to enable/disable high voltage detect feature for VDD_CORE in high power mode:

- **true** Enable low voltage detect feature for VDD_CORE in high power mode;
- **false** Disable low voltage detect feature for VDD_CORE in high power mode.

static inline void SPC_EnableHighPowerModeSystemLowVoltageDetect(SPC_Type *base, bool enable)

Enable/disable low voltage detect for VDD_SYS in high power mode.

Parameters

- base – SPC peripheral base address.
- enable – Used to enable/disable low voltage detect feature for VDD_SYS in high power mode:
  - **true** Enable low voltage detect feature for VDD_SYS in high power mode;
  - **false** Disable low voltage detect feature for VDD_SYS in high power mode.

static inline void SPC_EnableHighPowerModeSystemHighVoltageDetect(SPC_Type *base, bool enable)

Enable/disable high voltage detect for VDD_SYS in high power mode.

Parameters

- base – SPC peripheral base address.
- enable – Used to enable/disable high voltage detect feature for VDD_SYS in high power mode:
  - **true** Enable high voltage detect feature for VDD_SYS in high power mode;
  - **false** Disable high voltage detect feature for VDD_SYS in high power mode.

static inline void SPC_EnableHighPowerModeIOLowVoltageDetect(SPC_Type *base, bool enable)

Enable/disable low voltage detect for VDD_IO_ABC in high power mode.

Parameters

- base – SPC peripheral base address.
- enable – Used to enable/disable low voltage detect feature for VDD_IO_ABC in high power mode:
  - **true** Enable low voltage detect feature for VDD_IO_ABC in high power mode;
  - **false** Disable low voltage detect feature for VDD_IO_ABC in high power mode.

static inline void SPC_EnableHighPowerModeIOHighVoltageDetect(SPC_Type *base, bool enable)

Enable/disable high voltage detect for VDD_IO_ABC in high power mode.

Parameters

- base – SPC peripheral base address.
- enable – Used to enable/disable high voltage detect feature for VDD_IO_ABC in high power mode:
  - **true** Enable high voltage detect feature for VDD_IO_ABC in high power mode;
  - **false** Disable high voltage detect feature for VDD_IO_ABC in high power mode.

static inline *spc_core_ldo_voltage_level_t* SPC_GetActiveModeCoreLDOVDDVoltageLevel(SPC_Type *base)

Gets CORE LDO VDD Regulator Voltage level.

This function returns the voltage level of CORE LDO Regulator in Active mode.

Parameters

- base – SPC peripheral base address.

Returns

Voltage level of CORE LDO in type of spc_core_ldo_voltage_level_t enumeration.

static inline *spc_bandgap_mode_t* SPC_GetActiveModeBandgapMode(SPC_Type *base)

Gets the Bandgap mode in Active mode.

Parameters

- base – SPC peripheral base address.

Returns

Bandgap mode in the type of spc_bandgap_mode_t enumeration.

static inline uint32_t SPC_GetActiveModeVoltageDetectStatus(SPC_Type *base)

Gets all voltage detectors status in Active mode.

Parameters

- base – SPC peripheral base address.

Returns

All voltage detectors status in Active mode.

void SPC_SetActiveModeIntegratedPowerSwitchConfig(SPC_Type *base, const *spc_intergrated_power_switch_config_t* *config)

Configs Integrated power switch in active mode.

Note: Legacy API and will be removed.

Parameters

- base – SPC peripheral base address.
- config – Pointer to spc_intergrated_power_switch_config_t pointer.

status_t SPC_SetActiveModeBandgapModeConfig(SPC_Type *base, *spc_bandgap_mode_t* mode)

Configs Bandgap mode in Active mode.

Note: In active mode, because CORELDO_VDD_DS is reserved and set to Normal, so it is impossible to disable Bandgap in active mode

Parameters

- base – SPC peripheral base address.
- mode – The Bandgap mode be selected.

Return values

- kStatus_SPC_BandgapModeWrong – The Bandgap can not be disabled in active mode.

- kStatus_Success – Config Bandgap mode in Active power mode successful.

static inline void SPC_EnableActiveModeCMPBandgapBuffer(SPC_Type *base, bool enable)

Enables/Disable the CMP Bandgap Buffer in Active mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable CMP Bandgap buffer. true - Enable Buffer Stored Reference voltage to CMP. false - Disable Buffer Stored Reference voltage to CMP.

static inline void SPC_SetActiveModeVoltageTrimDelay(SPC_Type *base, uint16_t delay)

Sets the delay when the regulators change voltage level in Active mode.

Parameters

- base – SPC peripheral base address.
- delay – The number of SPC timer clock cycles.

status_t SPC_SetActiveModeRegulatorsConfig(SPC_Type *base, const *spc_active_mode_regulators_config_t* *config)

Configs regulators in Active mode.

This function provides the method to config all on-chip regulators in active mode.

Parameters

- base – SPC peripheral base address.
- config – Pointer to spc_active_mode_regulators_config_t structure.

Return values

- kStatus_Success – Config regulators in Active power mode successful.
- kStatus_SPC_BandgapModeWrong – The bandgap mode setting in Active mode is wrong.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_CORELDOVoltageWrong – The selected voltage level in active mode is not allowed.
- kStatus_SPC_SYSLDOOverDriveVoltageFail – Fail to regulator to Over Drive Voltage.
- kStatus_SPC_SYSLDOLowDriveStrengthIgnore – Set driver strength to Low will be ignored.
- kStatus_SPC_DCDCLowDriveStrengthIgnore – Set driver strength to Low will be ignored.
- kStatus_Timeout – Timeout occurs while waiting completion.

static inline void SPC_DisableActiveModeVddCoreGlitchDetect(SPC_Type *base, bool disable)

Disable/Enable VDD Core Glitch Detect in Active mode.

Note: State of glitch detect disable feature will be ignored if bandgap is disabled and glitch detect hardware will be forced to OFF state.

Parameters

- base – SPC peripheral base address.

- disable – Used to disable/enable VDD Core Glitch detect feature.
  - **true** Disable VDD Core Low Voltage detect;
  - **false** Enable VDD Core Low Voltage detect.

void SPC_SetLowPowerModeIntegratedPowerSwitchConfig(SPC_Type *base, const *spc_intergrated_power_switch_config_t* *config)

Configs Integrated power switch in Low Power mode.

Note: Legacy API, will be removed.

Parameters

- base – SPC peripheral base address.
- config – Pointer to spc_intergrated_power_switch_config_t pointer.

static inline void SPC_EnableLowPowerModeVDDCWellBias(SPC_Type *base, bool enable)

Enables/Disables VDDC Well Bias in low power mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable the VDDC Well Bias. true - Enable Vddc Well Bias. false - Disable Vddc Well Bias.

static inline *spc_core_ldo_drive_strength_t* SPC_GetLowPowerCoreLDOVDDDriveStrength(SPC_Type *base)

Gets CORE LDO VDD Drive Strength for Low Power modes.

Parameters

- base – SPC peripheral base address.

Returns

The CORE LDO's VDD Drive Strength.

static inline *spc_core_ldo_voltage_level_t* SPC_GetLowPowerCoreLDOVDDVoltageLevel(SPC_Type *base)

Gets the CORE LDO VDD Regulator Voltage Level for Low Power modes.

Parameters

- base – SPC peripheral base address.

Returns

The CORE LDO VDD Regulator's voltage level.

static inline *spc_bandgap_mode_t* SPC_GetLowPowerModeBandgapMode(SPC_Type *base)

Gets the Bandgap mode in Low Power mode.

Parameters

- base – SPC peripheral base address.

Returns

Bandgap mode in the type of spc_bandgap_mode_t enumeration.

static inline uint32_t SPC_GetLowPowerModeVoltageDetectStatus(SPC_Type *base)

Gets the status of all voltage detectors in Low Power mode.

Parameters

- base – SPC peripheral base address.

Returns
The status of all voltage detectors in low power mode.

static inline void SPC_EnableLowPowerModeLowPowerIREF(SPC_Type *base, bool enable)

Enables/Disables Low Power IREF in low power modes.

This function enables/disables Low Power IREF. Low Power IREF can only get disabled in Deep power down mode. In other low power modes, the Low Power IREF is always enabled.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable Low Power IREF. true - Enable Low Power IREF for Low Power modes. false - Disable Low Power IREF for Deep Power Down mode.

status_t SPC_SetLowPowerModeBandgapmodeConfig(SPC_Type *base, *spc_bandgap_mode_t* mode)

Configs Bandgap mode in Low Power mode.

This function configs Bandgap mode in Low Power mode. IF user want to disable Bandgap while keeping any of the Regulator in Normal Driver Strength or if any of the High voltage detectors/Low voltage detectors are kept enabled, the Bandgap mode will be set as Bandgap Enabled with Buffer Disabled.

Note: This API shall be invoked following set HVDs/LVDs and regulators' driver strength.

Parameters

- base – SPC peripheral base address.
- mode – The Bandgap mode be selected.

Return values

- kStatus_SPC_BandgapModeWrong – The bandgap mode setting in Low Power mode is wrong.
- kStatus_Success – Config Bandgap mode in Low Power power mode successful.

static inline void SPC_EnableLowPowerModeCMPBandgapBufferMode(SPC_Type *base, bool enable)

Enables/Disables CMP Bandgap Buffer.

This function gates CMP bandgap buffer. CMP bandgap buffer is automatically disabled and turned off in Deep Power Down mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable CMP Bandgap buffer. true - Enable Buffer Stored Reference Voltage to CMP. false - Disable Buffer Stored Reference Voltage to CMP.

static inline void SPC_EnableLowPowerModeCoreVDDInternalVoltageScaling(SPC_Type *base, bool enable)

Enables/Disables CORE VDD IVS(Internal Voltage Scaling) in low power modes.

This function gates CORE VDD IVS. When enabled, the IVS regulator will scale the external input CORE VDD to a lower voltage level to reduce internal leakage. IVS is invalid in Sleep or Deep power down mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable IVS. true - enable CORE VDD IVS in Deep Sleep mode or Power Down mode. false - disable CORE VDD IVS in Deep Sleep mode or Power Down mode.

static inline void SPC_SetLowPowerWakeUpDelay(SPC_Type *base, uint16_t delay)

Sets the delay when exit the low power modes.

Parameters

- base – SPC peripheral base address.
- delay – The number of SPC timer clock cycles that the SPC waits on exit from low power modes.

status_t SPC_SetLowPowerModeRegulatorsConfig(SPC_Type *base, const *spc_lowpower_mode_regulators_config_t* *config)

Configs regulators in Low Power mode.

This function provides the method to config all on-chip regulators in Low Power mode.

Parameters

- base – SPC peripheral base address.
- config – Pointer to spc_lowpower_mode_regulators_config_t structure.

Return values

- kStatus_Success – Config regulators in Low power mode successful.
- kStatus_SPC_BandgapModeWrong – The bandgap mode setting in Low Power mode is wrong.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_CORELDOVoltageWrong – The selected voltage level is wrong.
- kStatus_SPC_CORELDOLowDriveStrengthIgnore – Set driver strength to low will be ignored.
- kStatus_SPC_CORELDOVoltageSetFail – Fail to change Core LDO voltage level.
- kStatus_SPC_SYSLDOLowDriveStrengthIgnore – Set driver strength to low will be ignored.
- kStatus_SPC_DCDCPulseRefreshModeIgnore – Set driver strength to Pulse Refresh mode will be ignored.
- kStatus_SPC_DCDCLowDriveStrengthIgnore – Set driver strength to Low Drive Strength will be ignored.

static inline void SPC_DisableLowPowerModeVddCoreGlitchDetect(SPC_Type *base, bool disable)

Disable/Enable VDD Core Glitch Detect in low power mode.

Note: State of glitch detect disable feature will be ignored if bandgap is disabled and glitch detect hardware will be forced to OFF state.

Parameters

- base – SPC peripheral base address.
- disable – Used to disable/enable VDD Core Glitch detect feature.
  - **true** Disable VDD Core Low Voltage detect;
  - **false** Enable VDD Core Low Voltage detect.

static inline uint8_t SPC_GetVoltageDetectStatusFlag(SPC_Type *base)

Get Voltage Detect Status Flags.

Parameters

- base – SPC peripheral base address.

Returns

Voltage Detect Status Flags. See _spc_voltage_detect_flags for details.

static inline void SPC_ClearVoltageDetectStatusFlag(SPC_Type *base, uint8_t mask)

Clear Voltage Detect Status Flags.

Parameters

- base – SPC peripheral base address.
- mask – The mask of the voltage detect status flags. See _spc_voltage_detect_flags for details.

void SPC_SetCoreVoltageDetectConfig(SPC_Type *base, const *spc_core_voltage_detect_config_t* *config)

Configs CORE voltage detect options.

This function config CORE voltage detect options.

Note: : Setting both the voltage detect interrupt and reset enable will cause interrupt to be generated on exit from reset. If those conditioned is not desired, interrupt/reset so only one is enabled.

Parameters

- base – SPC peripheral base address.
- config – Pointer to spc_core_voltage_detect_config_t structure.

static inline void SPC_LockCoreVoltageDetectResetSetting(SPC_Type *base)

Locks Core voltage detect reset setting.

This function locks core voltage detect reset setting. After invoking this function any configuration of Core voltage detect reset will be ignored.

Parameters

- base – SPC peripheral base address.

static inline void SPC_UnlockCoreVoltageDetectResetSetting(SPC_Type *base)

Unlocks Core voltage detect reset setting.

This function unlocks core voltage detect reset setting. If locks the Core voltage detect reset setting, invoking this function to unlock.

Parameters

- base – SPC peripheral base address.

status_t SPC_EnableActiveModeCoreHighVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the Core High Voltage Detector in Active mode.

Note: If the CORE_LDO high voltage detect is enabled in Active mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable Core HVD. true - Enable Core High voltage detector in active mode. false - Disable Core High voltage detector in active mode.

Return values

kStatus_Success – Enable/Disable Core High Voltage Detect successfully.

status_t SPC_EnableActiveModeCoreLowVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the Core Low Voltage Detector in Active mode.

Note: If the CORE_LDO low voltage detect is enabled in Active mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable Core LVD. true - Enable Core Low voltage detector in active mode. false - Disable Core Low voltage detector in active mode.

Return values

kStatus_Success – Enable/Disable Core Low Voltage Detect successfully.

status_t SPC_EnableLowPowerModeCoreHighVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the Core High Voltage Detector in Low Power mode.

This function enables/disables the Core High Voltage Detector. If enabled the Core High Voltage detector. The Bandgap mode in low power mode must be programmed so that Bandgap is enabled.

Note: If the CORE_LDO high voltage detect is enabled in Low Power mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in low power mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable Core HVD. true - Enable Core High voltage detector in low power mode. false - Disable Core High voltage detector in low power mode.

Return values

kStatus_Success – Enable/Disable Core High Voltage Detect in low power mode successfully.

status_t SPC_EnableLowPowerModeCoreLowVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the Core Low Voltage Detector in Low Power mode.

This function enables/disables the Core Low Voltage Detector. If enabled the Core Low Voltage detector. The Bandgap mode in low power mode must be programmed so that Bandgap is enabled.

Note: If the CORE_LDO low voltage detect is enabled in Low Power mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Low Power mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable Core HVD. true - Enable Core Low voltage detector in low power mode. false - Disable Core Low voltage detector in low power mode.

Return values

kStatus_Success – Enable/Disable Core Low Voltage Detect in low power mode successfully.

void SPC_SetSystemVDDLowVoltageLevel(SPC_Type *base, *spc_low_voltage_level_select_t* level)

Set system VDD Low-voltage level selection.

This function selects the system VDD low-voltage level. Changing system VDD low-voltage level must be done after disabling the System VDD low voltage reset and interrupt.

Deprecated:

In latest RM, reserved for all devices, will removed in next release.

Parameters

- base – SPC peripheral base address.
- level – System VDD Low-Voltage level selection.

void SPC_SetSystemVoltageDetectConfig(SPC_Type *base, const *spc_system_voltage_detect_config_t* *config)

Configs SYS voltage detect options.

This function config SYS voltage detect options.

Note: : Setting both the voltage detect interrupt and reset enable will cause interrupt to be generated on exit from reset. If those conditioned is not desired, interrupt/reset so only one is enabled.

Parameters

- base – SPC peripheral base address.
- config – Pointer to spc_system_voltage_detect_config_t structure.

static inline void SPC_LockSystemVoltageDetectResetSetting(SPC_Type *base)

Lock System voltage detect reset setting.

This function locks system voltage detect reset setting. After invoking this function any configuration of System Voltage detect reset will be ignored.

Parameters

- base – SPC peripheral base address.

static inline void SPC_UnlockSystemVoltageDetectResetSetting(SPC_Type *base)

Unlock System voltage detect reset setting.

This function unlocks system voltage detect reset setting. If locks the System voltage detect reset setting, invoking this function to unlock.

Parameters

- base – SPC peripheral base address.

status_t SPC_EnableActiveModeSystemHighVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the System High Voltage Detector in Active mode.

Note: If the System_LDO high voltage detect is enabled in Active mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Active mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable System HVD. true - Enable System High voltage detector in active mode. false - Disable System High voltage detector in active mode.

Return values

kStatus_Success – Enable/Disable System High Voltage Detect successfully.

status_t SPC_EnableActiveModeSystemLowVoltageDetect(SPC_Type *base, bool enable)

Enables/Disable the System Low Voltage Detector in Active mode.

Note: If the System_LDO low voltage detect is enabled in Active mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Active mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable System LVD. true - Enable System Low voltage detector in active mode. false - Disable System Low voltage detector in active mode.

Return values

kStatus_Success – Enable/Disable the System Low Voltage Detect successfully.

status_t SPC_EnableLowPowerModeSystemHighVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the System High Voltage Detector in Low Power mode.

Note: If the System_LDO high voltage detect is enabled in Low Power mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Low Power mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable System HVD. true - Enable System High voltage detector in low power mode. false - Disable System High voltage detector in low power mode.

Return values

kStatus_Success – Enable/Disable System High Voltage Detect in low power mode successfully.

status_t SPC_EnableLowPowerModeSystemLowVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the System Low Voltage Detector in Low Power mode.

Note: If the System_LDO low voltage detect is enabled in Low Power mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Low Power mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable System HVD. true - Enable System Low voltage detector in low power mode. false - Disable System Low voltage detector in low power mode.

Return values

kStatus_Success – Enables System Low Voltage Detect in low power mode successfully.

void SPC_SetIOVDDLowVoltageLevel(SPC_Type *base, *spc_low_voltage_level_select_t* level)

Set IO VDD Low-Voltage level selection.

This function selects the IO VDD Low-voltage level. Changing IO VDD low-voltage level must be done after disabling the IO VDD low voltage reset and interrupt.

Parameters

- base – SPC peripheral base address.
- level – IO VDD Low-voltage level selection.

void SPC_SetIOVoltageDetectConfig(SPC_Type *base, const *spc_io_voltage_detect_config_t* *config)

Configs IO voltage detect options.

This function config IO voltage detect options.

Note: : Setting both the voltage detect interrupt and reset enable will cause interrupt to be generated on exit from reset. If those conditioned is not desired, interrupt/reset so only one is enabled.

Parameters

- base – SPC peripheral base address.
- config – Pointer to spc_voltage_detect_config_t structure.

static inline void SPC_LockIOVoltageDetectResetSetting(SPC_Type *base)

Lock IO Voltage detect reset setting.

This function locks IO voltage detect reset setting. After invoking this function any configuration of system voltage detect reset will be ignored.

Parameters

- base – SPC peripheral base address.

static inline void SPC_UnlockIOVoltageDetectResetSetting(SPC_Type *base)

Unlock IO voltage detect reset setting.

This function unlocks IO voltage detect reset setting. If locks the IO voltage detect reset setting, invoking this function to unlock.

Parameters

- base – SPC peripheral base address.

status_t SPC_EnableActiveModeIOHighVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the IO High Voltage Detector in Active mode.

Note: If the IO high voltage detect is enabled in Active mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Active mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable IO HVD. true - Enable IO High voltage detector in active mode. false - Disable IO High voltage detector in active mode.

Return values

kStatus_Success – Enable/Disable IO High Voltage Detect successfully.

status_t SPC_EnableActiveModeIOLowVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the IO Low Voltage Detector in Active mode.

Note: If the IO low voltage detect is enabled in Active mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Active mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable IO LVD. true - Enable IO Low voltage detector in active mode. false - Disable IO Low voltage detector in active mode.

Return values

kStatus_Success – Enable IO Low Voltage Detect successfully.

status_t SPC_EnableLowPowerModeIOHighVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the IO High Voltage Detector in Low Power mode.

Note: If the IO high voltage detect is enabled in Low Power mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Low Power mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable IO HVD. true - Enable IO High voltage detector in low power mode. false - Disable IO High voltage detector in low power mode.

Return values

kStatus_Success – Enable IO High Voltage Detect in low power mode successfully.

status_t SPC_EnableLowPowerModeIOLowVoltageDetect(SPC_Type *base, bool enable)

Enables/Disables the IO Low Voltage Detector in Low Power mode.

Note: If the IO low voltage detect is enabled in Low Power mode, please note that the bandgap must be enabled and the drive strength of each regulator must not set to low in Low Power mode.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable IO LVD. true - Enable IO Low voltage detector in low power mode. false - Disable IO Low voltage detector in low power mode.

Return values

kStatus_Success – Enable/Disable IO Low Voltage Detect in low power mode successfully.

void SPC_SetExternalVoltageDomainsConfig(SPC_Type *base, uint8_t lowPowerIsoMask, uint8_t IsoMask)

Configs external voltage domains.

This function configs external voltage domains isolation.

Parameters

- base – SPC peripheral base address.
- lowPowerIsoMask – The mask of external domains isolate enable during low power mode. Please read the Reference Manual for the Bitmap.
- IsoMask – The mask of external domains isolate. Please read the Reference Manual for the Bitmap.

static inline uint8_t SPC_GetExternalDomainsStatus(SPC_Type *base)

Gets External Domains status.

This function configs external voltage domains status.

Parameters

- base – SPC peripheral base address.

Returns

The status of each external domain.

static inline void SPC_EnableCoreLDORegulator(SPC_Type *base, bool enable)

Enable/Disable Core LDO regulator.

Note: The CORE LDO enable bit is write-once.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable CORE LDO Regulator. true - Enable CORE LDO Regulator. false - Disable CORE LDO Regulator.

static inline void SPC_PullDownCoreLDORegulator(SPC_Type *base, bool pulldown)

Enable/Disable the CORE LDO Regulator pull down in Deep Power Down.

Note: This function only useful when enabled the CORE LDO Regulator.

Parameters

- base – SPC peripheral base address.
- pulldown – Enable/Disable CORE LDO pulldown in Deep Power Down mode. true - CORE LDO Regulator will discharge in Deep Power Down mode. false - CORE LDO Regulator will not discharge in Deep Power Down mode.

status_t SPC_SetActiveModeCoreLDORegulatorConfig(SPC_Type *base, const *spc_active_mode_core_ldo_option_t* *option)

Configs Core LDO VDD Regulator in Active mode.

Note: If any voltage detect feature is enabled in Active mode, then CORE_LDO's drive strength must not set to low.

Note: Core VDD level for the Core LDO low power regulator can only be changed when CORELDO_VDD_DS is normal

Parameters

- base – SPC peripheral base address.
- option – Pointer to the spc_active_mode_core_ldo_option_t structure.

Return values

- kStatus_Success – Config Core LDO regulator in Active power mode successful.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_CORELDOLowDriveStrengthIgnore – If any voltage detect enabled, core_ldo's drive strength can not set to low.
- kStatus_SPC_CORELDOVoltageWrong – The selected voltage level in active mode is not allowed.
- kStatus_Timeout – Timeout occurs while waiting completion.

status_t SPC_SetLowPowerModeCoreLDORegulatorConfig(SPC_Type *base, const *spc_lowpower_mode_core_ldo_option_t* *option)

Configs CORE LDO Regulator in low power mode.

This function configs CORE LDO Regulator in Low Power mode. If CORE LDO VDD Drive Strength is set to Normal, the CORE LDO VDD regulator voltage level in Active mode must be equal to the voltage level in Low power mode. And the Bandgap must be programmed to select bandgap enabled. Core VDD voltage levels for the Core LDO low power regulator can only be changed when the CORE LDO Drive Strength set as Normal.

Parameters

- base – SPC peripheral base address.

- option – Pointer to the spc_lowpower_mode_core_ldo_option_t structure.

Return values

- kStatus_Success – Config Core LDO regulator in power mode successfully.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_CORELDOLowDriveStrengthIgnore – Set driver strength to low will be ignored.
- kStatus_SPC_CORELDOVoltageSetFail – Fail to change Core LDO voltage level.
- kStatus_Timeout – Timeout occurs while waiting completion.

static inline void SPC_EnableSystemLDORegulator(SPC_Type *base, bool enable)

Enable/Disable System LDO regulator.

Note: The SYSTEM LDO enable bit is write-once.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable System LDO Regulator. true - Enable System LDO Regulator. false - Disable System LDO Regulator.

static inline void SPC_EnableSystemLDOSinkFeature(SPC_Type *base, bool sink)

Enable/Disable current sink feature of System LDO Regulator.

Parameters

- base – SPC peripheral base address.
- sink – Enable/Disable current sink feature. true - Enable current sink feature of System LDO Regulator. false - Disable current sink feature of System LDO Regulator.

status_t SPC_SetActiveModeSystemLDORegulatorConfig(SPC_Type *base, const *spc_active_mode_sys_ldo_option_t* *option)

Configs System LDO VDD Regulator in Active mode.

This function configs System LDO VDD Regulator in Active mode. If System LDO VDD Drive Strength is set to Normal, the Bandgap mode in Active mode must be programmed to a value that enables the bandgap. If any voltage detects are kept enabled, configuration to set System LDO VDD drive strength to low will be ignored. If select System LDO VDD Regulator voltage level to Over Drive Voltage, the Drive Strength of System LDO VDD Regulator must be set to Normal otherwise the regulator Drive Strength will be forced to Normal. If select System LDO VDD Regulator voltage level to Over Drive Voltage, the High voltage detect must be disabled. Otherwise it will be fail to regulator to Over Drive Voltage.

Parameters

- base – SPC peripheral base address.
- option – Pointer to the spc_active_mode_sys_ldo_option_t structure.

Return values

- kStatus_Success – Config System LDO regulator in Active power mode successful.

- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_SYSLDOOverDriveVoltageFail – Fail to regulator to Over Drive Voltage.
- kStatus_SPC_SYSLDOLowDriveStrengthIgnore – Set driver strength to Low will be ignored.
- kStatus_Timeout – Timeout occurs while waiting completion.

status_t SPC_SetLowPowerModeSystemLDORegulatorConfig(SPC_Type *base, const *spc_lowpower_mode_sys_ldo_option_t* *option)

Configs System LDO regulator in low power modes.

This function configs System LDO regulator in low power modes. If System LDO VDD Regulator Drive strength is set to normal, bandgap mode in low power mode must be programmed to a value that enables the Bandgap. If any High voltage detectors or Low Voltage detectors are kept enabled, configuration to set System LDO Regulator drive strength as Low will be ignored.

Parameters

- base – SPC peripheral base address.
- option – Pointer to spc_lowpower_mode_sys_ldo_option_t structure.

Return values

- kStatus_Success – Config System LDO regulator in Low Power Mode successfully.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_SYSLDOLowDriveStrengthIgnore – Set driver strength to low will be ignored.
- kStatus_Timeout – Timeout occurs while waiting completion.

static inline void SPC_EnableDCDCRegulator(SPC_Type *base, bool enable)

Enable/Disable DCDC Regulator.

Note: The DCDC enable bit is write-once.

Parameters

- base – SPC peripheral base address.
- enable – Enable/Disable DCDC Regulator. true - Enable DCDC Regulator. false - Disable DCDC Regulator.

status_t SPC_SetDCDCBurstConfig(SPC_Type *base, *spc_dcdc_burst_config_t* *config)

Config DCDC Burst options.

Parameters

- base – SPC peripheral base address.
- config – Pointer to spc_dcdc_burst_config_t structure.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

void SPC_SetDCDCRefreshCount(SPC_Type *base, uint16_t count)

Set the count value of the reference clock.

This function set the count value of the reference clock to control the frequency of dcdc refresh when dcdc is configured in Pulse Refresh mode.

Parameters

- base – SPC peripheral base address.
- count – The count value, 16 bit width.

status_t SPC_SetActiveModeDCDCRegulatorConfig(SPC_Type *base, const *spc_active_mode_dcdc_option_t* *option)

Configs DCDC VDD Regulator in Active mode.

This function configs DCDC VDD Regulator in Active mode. If DCDDC VDD Drive Strength is set to Normal, the Bandgap mode in Active mode must be programmed to a value that enable the bandgap. If any voltage detects are kept enabled, configuration to set DCDC VDD drive strength to low will be ignored. When switching DCDC from low drive strength to Normal driver strength, make sure the DCDC high VDD LVL setting to the same level that was set prior to switching the DCDC to low drive strength.

Parameters

- base – SPC peripheral base address.
- option – Pointer to the spc_active_mode_dcdc_option_t structure.

Return values

- kStatus_Success – Config DCDC regulator in Active power mode successful.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_DCDCLowDriveStrengthIgnore – Set driver strength to Low will be ignored.
- kStatus_Timeout – Timeout occurs while waiting completion.

status_t SPC_SetLowPowerModeDCDCRegulatorConfig(SPC_Type *base, const *spc_lowpower_mode_dcdc_option_t* *option)

Configs DCDC VDD Regulator in Low power modes.

This function configs DCDC VDD Regulator in Low Power modes. If DCDC VDD Drive Strength is set to Normal, the Bandgap mode in Low Power mode must be programmed to a value that enables the Bandgap. If any of voltage detectors are kept enabled, configuration to set DCDC VDD Drive Strength to Low or Pulse mode will be ignored. In Deep Power Down mode, DCDC regulator is always turned off.

Parameters

- base – SPC peripheral base address.
- option – Pointer to the spc_lowpower_mode_dcdc_option_t structure.

Return values

- kStatus_Success – Config DCDC regulator in low power mode successfully.
- kStatus_SPC_Busy – The SPC instance is busy to execute any type of power mode transition.
- kStatus_SPC_DCDCPulseRefreshModeIgnore – Set driver strength to Pulse Refresh mode will be ignored.

- kStatus_SPC_DCDCLowDriveStrengthIgnore – Set driver strength to Low Drive Strength will be ignored.
- kStatus_Timeout – Timeout occurs while waiting completion.

status_t SPC_SetSRAMOperateVoltage(SPC_Type *base, *spc_sram_operat_voltage_t* voltage)

Set the SRAM operate voltage level.

Parameters

- base – SPC peripheral base address.
- voltage – Target SRAM operate voltage level, please refer to spc_sram_operat_voltage_t.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

void SPC_ConfigVddCoreGlitchDetector(SPC_Type *base, const *spc_vdd_core_glitch_detector_config_t* *config)

Configures VDD Core Glitch detector, including ripple counter selection, timeout value and so on.

Parameters

- base – SPC peripheral base address.
- config – Pointer to the structure in type of spc_vdd_core_glitch_detector_config_t.

static inline bool SPC_CheckGlitchRippleCounterOutput(SPC_Type *base, *spc_vdd_core_glitch_ripple_counter_select_t* rippleCounter)

Checks selected 4-bit glitch ripple counter's output.

Parameters

- base – SPC peripheral base address.
- rippleCounter – The ripple counter to check, please refer to spc_vdd_core_glitch_ripple_counter_select_t.

Return values

- true – The selected ripple counter output is 1, will generate interrupt or reset based on settings.
- false – The selected ripple counter output is 0.

static inline void SPC_ClearGlitchRippleCounterOutput(SPC_Type *base, *spc_vdd_core_glitch_ripple_counter_select_t* rippleCounter)

Clears output of selected glitch ripple counter.

Parameters

- base – SPC peripheral base address.
- rippleCounter – The ripple counter to check, please refer to spc_vdd_core_glitch_ripple_counter_select_t.

static inline void SPC_LockVddCoreVoltageGlitchDetectResetControl(SPC_Type *base)

After invoking this function, writes to SPC_VDD_CORE_GLITCH_DETECT_SC[RE] register are ignored.

Parameters

- base – SPC peripheral base address.

static inline void SPC_UnlockVddCoreVoltageGlitchDetectResetControl(SPC_Type *base)

After invoking this function, writes to SPC_VDD_CORE_GLITCH_DETECT_SC[RE] register are allowed.

Parameters

- base – SPC peripheral base address.

static inline bool SPC_CheckVddCoreVoltageGlitchResetControlState(SPC_Type *base)

Checks if SPC_VDD_CORE_GLITCH_DETECT_SC[RE] register is writable.

Parameters

- base – SPC peripheral base address.

Return values

- true – SPC_VDD_CORE_GLITCH_DETECT_SC[RE] register is writable.
- false – SPC_VDD_CORE_GLITCH_DETECT_SC[RE] register is not writable.

FSL_SPC_DRIVER_VERSION

SPC driver version 2.8.0.

SPC_EVD_CFG_REG_EVDISO_SHIFT

SPC_EVD_CFG_REG_EVDLPISO_SHIFT

SPC_EVD_CFG_REG_EVDSTAT_SHIFT

SPC_EVD_CFG_REG_EVDISO(x)

SPC_EVD_CFG_REG_EVDLPISO(x)

SPC_EVD_CFG_REG_EVDSTAT(x)

struct _spc_lowpower_request_config

#include <fsl_spc.h> Low Power Request output pin configuration.

Public Members

bool enable

Low Power Request Output enable.

spc_lowpower_request_pin_polarity_t polarity

Low Power Request Output pin polarity select.

spc_lowpower_request_output_override_t override

Low Power Request Output Override.

struct _spc_intergrated_power_switch_config

#include <fsl_spc.h> Integrated power switch configuration.

Note: Legacy structure, will be removed.

Public Members

bool wakeup

Assert an output pin to un-gate the integrated power switch.

bool sleep
Assert an output pin to power gate the intergrated power switch.

struct __spc_active_mode_core_ldo_option
#include <fsl_spc.h> Core LDO regulator options in Active mode.

Public Members

spc_core_ldo_voltage_level_t CoreLDOVoltage
Core LDO Regulator Voltage Level selection in Active mode.

spc_core_ldo_drive_strength_t CoreLDODriveStrength
Core LDO Regulator Drive Strength selection in Active mode

struct __spc_active_mode_sys_ldo_option
#include <fsl_spc.h> System LDO regulator options in Active mode.

Public Members

spc_sys_ldo_voltage_level_t SysLDOVoltage
System LDO Regulator Voltage Level selection in Active mode.

spc_sys_ldo_drive_strength_t SysLDODriveStrength
System LDO Regulator Drive Strength selection in Active mode.

struct __spc_active_mode_dcdc_option
#include <fsl_spc.h> DCDC regulator options in Active mode.

Public Members

spc_dcdc_voltage_level_t DCDCVoltage
DCDC Regulator Voltage Level selection in Active mode.

spc_dcdc_drive_strength_t DCDCDriveStrength
DCDC VDD Regulator Drive Strength selection in Active mode.

struct __spc_lowpower_mode_core_ldo_option
#include <fsl_spc.h> Core LDO regulator options in Low Power mode.

Public Members

spc_core_ldo_voltage_level_t CoreLDOVoltage
Core LDO Regulator Voltage Level selection in Low Power mode.

spc_core_ldo_drive_strength_t CoreLDODriveStrength
Core LDO Regulator Drive Strength selection in Low Power mode

struct __spc_lowpower_mode_sys_ldo_option
#include <fsl_spc.h> System LDO regulator options in Low Power mode.

Public Members

spc_sys_ldo_drive_strength_t SysLDODriveStrength
System LDO Regulator Drive Strength selection in Low Power mode.

struct __spc_lowpower_mode_dcdc_option
#include <fsl_spc.h> DCDC regulator options in Low Power mode.

Public Members

spc_dcdc_voltage_level_t DCDCVoltage
DCDC Regulator Voltage Level selection in Low Power mode.

spc_dcdc_drive_strength_t DCDCDriveStrength
DCDC VDD Regulator Drive Strength selection in Low Power mode.

struct __spc__voltage__detect__option
#include <fsl_spc.h> CORE/SYS/IO VDD Voltage Detect options.

Public Members

bool HVDInterruptEnable
CORE/SYS/IO VDD High Voltage Detect interrupt enable.

bool HVDResetEnable
CORE/SYS/IO VDD High Voltage Detect reset enable.

bool LVDInterruptEnable
CORE/SYS/IO VDD Low Voltage Detect interrupt enable.

bool LVDResetEnable
CORE/SYS/IO VDD Low Voltage Detect reset enable.

struct __spc__dcdc__burst__config
#include <fsl_spc.h> DCDC Burst configuration.

Public Members

bool sofwareBurstRequest
Enable/Disable DCDC Software Burst Request.

bool externalBurstRequest
Enable/Disable DCDC External Burst Request.

bool stabilizeBurstFreq
Enable/Disable DCDC frequency stabilization.

uint8_t freq
The frequency of the current burst.

struct __spc__core__voltage__detect__config
#include <fsl_spc.h> Core Voltage Detect configuration.

Public Members

spc_voltage_detect_option_t option
Core VDD Voltage Detect option.

struct __spc__system__voltage__detect__config
#include <fsl_spc.h> System Voltage Detect Configuration.

Public Members

spc_voltage_detect_option_t option
System VDD Voltage Detect option.

spc_low_voltage_level_select_t level

Deprecated:

, reserved for all devices, will removed in next release.

struct __spc__io__voltage__detect__config

#include <fsl_spc.h> IO Voltage Detect Configuration.

Public Members

spc_voltage_detect_option_t option

IO VDD Voltage Detect option.

spc_low_voltage_level_select_t level

IO VDD Low-voltage level selection.

struct __spc__active__mode__regulators__config

#include <fsl_spc.h> Active mode configuration.

struct __spc__lowpower__mode__regulators__config

#include <fsl_spc.h> Low Power Mode configuration.

struct __spc__vdd__core__glitch__detector__config

#include <fsl_spc.h> The configuration of VDD Core glitch detector.

Public Members

spc_vdd_core_glitch_ripple_counter_select_t rippleCounterSelect

Used to set ripple counter.

uint8_t resetTimeoutValue

The timeout value used to reset glitch detect/compare logic after an initial glitch is detected.

bool enableReset

Used to enable/disable POR/LVD reset that caused by CORE VDD glitch detect error.

bool enableInterrupt

Used to enable/disable hardware interrupt if CORE VDD glitch detect error.

2.61 SYSPM: System Performance Monitor

enum __syspm__monitor

syspm select control monitor

Values:

enumerator kSYSPM__Monitor0

Monitor 0

enum __syspm__event

syspm select event

Values:

enumerator kSYSPM__Event1

Event 1

enumerator kSYSPM_Event2
Event 2

enumerator kSYSPM_Event3
Event 3

enum _syspm_mode
syspm set count mode

Values:

enumerator kSYSPM_BothMode
count in both modes

enumerator kSYSPM_UserMode
count only in user mode

enumerator kSYSPM_PrivilegedMode
count only in privileged mode

enum _syspm_startstop_control
syspm start/stop control

Values:

enumerator kSYSPM_Idle
idle >

enumerator kSYSPM_LocalStop
local stop

enumerator kSYSPM_LocalStart
local start

enumerator KSYSPM_EnableTraceControl
enable global TSTART/TSTOP

enumerator kSYSPM_GlobalStart
global stop

enumerator kSYSPM_GlobalStop
global start

typedef enum *_syspm_monitor* syspm_monitor_t
syspm select control monitor

typedef enum *_syspm_event* syspm_event_t
syspm select event

typedef enum *_syspm_mode* syspm_mode_t
syspm set count mode

typedef enum *_syspm_startstop_control* syspm_startstop_control_t
syspm start/stop control

void SYSPM_Init(SYSPM_Type *base)
Initializes the SYSPM.

This function enables the SYSPM clock.

Parameters

- base – SYSPM peripheral base address.

void SYSPM_Deinit(SYSPM_Type *base)

Deinitializes the SYSPM.

This function disables the SYSPM clock.

Parameters

- base – SYSPM peripheral base address.

void SYSPM_SelectEvent(SYSPM_Type *base, *syspm_monitor_t* monitor, *syspm_event_t* event, uint8_t eventCode)

Select event counters.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to syspm_monitor_t.
- event – syspm select event, see to syspm_event_t.
- eventCode – select which event to be counted in PMECTRx., see to table Events.

void SYSPM_ResetEvent(SYSPM_Type *base, *syspm_monitor_t* monitor, *syspm_event_t* event)

Reset event counters.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to syspm_monitor_t.
- event – syspm select event, see to syspm_event_t.

void SYSPM_ResetInstructionEvent(SYSPM_Type *base, *syspm_monitor_t* monitor)

Reset Instruction Counter.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to syspm_monitor_t.

void SYSPM_SetCountMode(SYSPM_Type *base, *syspm_monitor_t* monitor, *syspm_mode_t* mode)

Set count mode.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to syspm_monitor_t.
- mode – syspm select counter mode, see to syspm_mode_t.

void SYSPM_SetStartStopControl(SYSPM_Type *base, *syspm_monitor_t* monitor, *syspm_startstop_control_t* ssc)

Set Start/Stop Control.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to syspm_monitor_t.
- ssc – This 3-bit field provides a three-phase mechanism to start/stop the counters. It includes a prioritized scheme with local start > local stop > global start > global stop > conditional TSTART > TSTOP. The global and conditional start/stop affect all configured PM/PSAM module concurrently so counters are “coherent”. see to syspm_startstop_control_t

void SYSPM_DisableCounter(SYSPM_Type *base, *syspm_monitor_t* monitor)

Disable Counters if Stopped or Halted.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to syspm_monitor_t.

uint64_t SYSPM_GetEventCounter(SYSPM_Type *base, *syspm_monitor_t* monitor, *syspm_event_t* event)

This is the the 40-bits of eventx counter. The value in this register increments each time the event selected in PMCRx[SELEVTx] occurs.

Parameters

- base – SYSPM peripheral base address.
- monitor – syspm control monitor, see to syspm_monitor_t.
- event – syspm select event, see to syspm_event_t.

Returns

- When the return value is not equal to SYSPM_COUNT_STABLE_TIMEOUT_RETURN_VALUE, the return value represents a 40 bits eventx counter.
- When the return value is equal to SYSPM_COUNT_STABLE_TIMEOUT_RETURN_VALUE, the return value represents timeout occured.

EVENT_COUNT_STABLE_TIMEOUT

Max loops to wait for SYSPM event count stable (0 means wait forever)

INSTRUCTION_COUNT_STABLE_TIMEOUT

Max loops to wait for SYSPM instruction count stable (0 means wait forever)

FSL_SYSPM_DRIVER_VERSION

SYSPM driver version.

SYSPM_COUNT_STABLE_TIMEOUT_RETURN_VALUE

2.62 TPM: Timer PWM Module

uint32_t TPM_GetInstance(TPM_Type *base)

Gets the instance from the base address.

Parameters

- base – TPM peripheral base address

Returns

The TPM instance

void TPM_Init(TPM_Type *base, const *tpm_config_t* *config)

Ungates the TPM clock and configures the peripheral for basic operation.

Note: This API should be called at the beginning of the application using the TPM driver.

Parameters

- base – TPM peripheral base address

- config – Pointer to user's TPM config structure.

void TPM_Deinit(TPM_Type *base)

Stops the counter and gates the TPM clock.

Parameters

- base – TPM peripheral base address

void TPM_GetDefaultConfig(*tpm_config_t* *config)

Fill in the TPM config struct with the default settings.

The default values are:

```
    config->prescale = kTPM_Prescale_Divide_1;
    config->useGlobalTimeBase = false;
    config->syncGlobalTimeBase = false;
    config->dozeEnable = false;
    config->dbgMode = false;
    config->enableReloadOnTrigger = false;
    config->enableStopOnOverflow = false;
    config->enableStartOnTrigger = false;
#if FSL_FEATURE_TPM_HAS_PAUSE_COUNTER_ON_TRIGGER
    config->enablePauseOnTrigger = false;
#endif
    config->triggerSelect = kTPM_Trigger_Select_0;
#if FSL_FEATURE_TPM_HAS_EXTERNAL_TRIGGER_SELECTION
    config->triggerSource = kTPM_TriggerSource_External;
    config->extTriggerPolarity = kTPM_ExtTrigger_Active_High;
#endif
#if defined(FSL_FEATURE_TPM_HAS_POL) && FSL_FEATURE_TPM_HAS_POL
    config->chnlPolarity = 0U;
#endif
```

Parameters

- config – Pointer to user's TPM config structure.

tpm_clock_prescale_t TPM_CalculateCounterClkDiv(TPM_Type *base, uint32_t counterPeriod_Hz, uint32_t srcClock_Hz)

Calculates the counter clock prescaler.

This function calculates the values for SC[PS].

return Calculated clock prescaler value.

Parameters

- base – TPM peripheral base address
- counterPeriod_Hz – The desired frequency in Hz which corresponding to the time when the counter reaches the mod value
- srcClock_Hz – TPM counter clock in Hz

static inline void TPM_Reset(TPM_Type *base)

Performs a software reset on the TPM module.

Reset all internal logic and registers, except the Global Register. Remains set until cleared by software.

Note: TPM software reset is available on certain SoC's only

Parameters

- base – TPM peripheral base address

status_t TPM_SetupPwm(TPM_Type *base, const *tpm_chnl_pwm_signal_param_t* *chnlParams, uint8_t numOfChnls, *tpm_pwm_mode_t* mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz)

Configures the PWM signal parameters.

User calls this function to configure the PWM signals period, mode, dutycycle and edge. Use this function to configure all the TPM channels that will be used to output a PWM signal

Parameters

- base – TPM peripheral base address
- chnlParams – Array of PWM channel parameters to configure the channel(s)
- numOfChnls – Number of channels to configure, this should be the size of the array passed in
- mode – PWM operation mode, options available in enumeration tpm_pwm_mode_t
- pwmFreq_Hz – PWM signal frequency in Hz
- srcClock_Hz – TPM counter clock in Hz

Returns

kStatus_Success PWM setup successful kStatus_Error PWM setup failed kStatus_Timeout PWM setup timeout when write register CnV or MOD

status_t TPM_UpdatePwmDutycycle(TPM_Type *base, *tpm_chnl_t* chnlNumber, *tpm_pwm_mode_t* currentPwmMode, uint8_t dutyCyclePercent)

Update the duty cycle of an active PWM signal.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number. In combined mode, this represents the channel pair number
- currentPwmMode – The current PWM mode set during PWM setup
- dutyCyclePercent – New PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)

Returns

kStatus_Success if the PWM setup was successful, kStatus_Error on failure

void TPM_UpdateChnlEdgeLevelSelect(TPM_Type *base, *tpm_chnl_t* chnlNumber, uint8_t level)

Update the edge level selection for a channel.

Note: When the TPM has PWM pause level select feature (FSL_FEATURE_TPM_HAS_PAUSE_LEVEL_SELECT = 1), the PWM output cannot be turned off by selecting the output level. In this case, must use TPM_DisableChannel API to close the PWM output.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number

- level – The level to be set to the ELSnB:ELSnA field; valid values are 00, 01, 10, 11. See the appropriate SoC reference manual for details about this field.

static inline uint8_t TPM_GetChannelContorlBits(TPM_Type *base, *tpm_chnl_t* chnlNumber)

Get the channel control bits value (mode, edge and level bit fileds).

This function disable the channel by clear all mode and level control bits.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number

Returns

The contorl bits value. This is the logical OR of members of the enumeration tpm_chnl_control_bit_mask_t.

static inline *status_t* TPM_DisableChannel(TPM_Type *base, *tpm_chnl_t* chnlNumber)

Dsiable the channel.

This function disable the channel by clear all mode and level control bits.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number

Returns

kStatus_Success PWM setup successful kStatus_Timeout PWM setup timeout when write register CnSC

static inline *status_t* TPM_EnableChannel(TPM_Type *base, *tpm_chnl_t* chnlNumber, uint8_t control)

Enable the channel according to mode and level configs.

This function enable the channel output according to input mode/level config parameters.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- control – The contorl bits value. This is the logical OR of members of the enumeration tpm_chnl_control_bit_mask_t.

Returns

kStatus_Success PWM setup successful kStatus_Timeout PWM setup timeout when write register CnSC

void TPM_SetupInputCapture(TPM_Type *base, *tpm_chnl_t* chnlNumber, *tpm_input_capture_edge_t* captureMode)

Enables capturing an input signal on the channel using the function parameters.

When the edge specified in the captureMode argument occurs on the channel, the TPM counter is captured into the CnV register. The user has to read the CnV register separately to get this value.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- captureMode – Specifies which edge to capture

status_t TPM_SetupOutputCompare(TPM_Type *base, *tpm_chnl_t* chnlNumber, *tpm_output_compare_mode_t* compareMode, uint32_t compareValue)

Configures the TPM to generate timed pulses.

When the TPM counter matches the value of compareVal argument (this is written into CnV reg), the channel output is changed based on what is specified in the compareMode argument.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- compareMode – Action to take on the channel output when the compare condition is met
- compareValue – Value to be programmed in the CnV register.

Returns

kStatus_Success PWM setup successful kStatus_Timeout PWM setup timeout when write register CnV

void TPM_SetupDualEdgeCapture(TPM_Type *base, *tpm_chnl_t* chnlPairNumber, const *tpm_dual_edge_capture_param_t* *edgeParam, uint32_t filterValue)

Configures the dual edge capture mode of the TPM.

This function allows to measure a pulse width of the signal on the input of channel of a channel pair. The filter function is disabled if the filterVal argument passed is zero.

Parameters

- base – TPM peripheral base address
- chnlPairNumber – The TPM channel pair number; options are 0, 1, 2, 3
- edgeParam – Sets up the dual edge capture function
- filterValue – Filter value, specify 0 to disable filter.

void TPM_SetupQuadDecode(TPM_Type *base, const *tpm_phase_params_t* *phaseAParams, const *tpm_phase_params_t* *phaseBParams, *tpm_quad_decode_mode_t* quadMode)

Configures the parameters and activates the quadrature decode mode.

Parameters

- base – TPM peripheral base address
- phaseAParams – Phase A configuration parameters
- phaseBParams – Phase B configuration parameters
- quadMode – Selects encoding mode used in quadrature decoder mode

static inline void TPM_SetChannelPolarity(TPM_Type *base, *tpm_chnl_t* chnlNumber, bool enable)

Set the input and output polarity of each of the channels.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- enable – true: Set the channel polarity to active high; false: Set the channel polarity to active low;

static inline void TPM_EnableChannelExtTrigger(TPM_Type *base, *tpm_chnl_t* chnlNumber, bool enable)

Enable external trigger input to be used by channel.

In input capture mode, configures the trigger input that is used by the channel to capture the counter value. In output compare or PWM mode, configures the trigger input used to modulate the channel output. When modulating the output, the output is forced to the channel initial value whenever the trigger is not asserted.

Note: No matter how many external trigger sources there are, only input trigger 0 and 1 are used. The even numbered channels share the input trigger 0 and the odd numbered channels share the second input trigger 1.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number
- enable – true: Configures trigger input 0 or 1 to be used by channel; false: Trigger input has no effect on the channel

void TPM_EnableInterrupts(TPM_Type *base, uint32_t mask)

Enables the selected TPM interrupts.

Parameters

- base – TPM peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration tpm_interrupt_enable_t

void TPM_DisableInterrupts(TPM_Type *base, uint32_t mask)

Disables the selected TPM interrupts.

Parameters

- base – TPM peripheral base address
- mask – The interrupts to disable. This is a logical OR of members of the enumeration tpm_interrupt_enable_t

uint32_t TPM_GetEnabledInterrupts(TPM_Type *base)

Gets the enabled TPM interrupts.

Parameters

- base – TPM peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration tpm_interrupt_enable_t

void TPM_RegisterCallBack(TPM_Type *base, *tpm_callback_t* callback)

Register callback.

If channel or overflow interrupt is enabled by the user, then a callback can be registered which will be invoked when the interrupt is triggered.

Parameters

- base – TPM peripheral base address
- callback – Callback function

void TPM_DriverIRQHandler(uint32_t instance)

TPM driver IRQ handler common entry.

This function provides the common IRQ request entry for TPM.

Parameters

- instance – TPM instance.

static inline uint32_t TPM_GetChannelValue(TPM_Type *base, *tpm_chnl_t* chnlNumber)

Gets the TPM channel value.

Note: The TPM channel value contain the captured TPM counter value for the input modes or the match value for the output modes.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number

Returns

The channle CnV regisyer value.

static inline uint32_t TPM_GetStatusFlags(TPM_Type *base)

Gets the TPM status flags.

Parameters

- base – TPM peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration tpm_status_flags_t

static inline void TPM_ClearStatusFlags(TPM_Type *base, uint32_t mask)

Clears the TPM status flags.

Parameters

- base – TPM peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration tpm_status_flags_t

static inline *status_t* TPM_SetTimerPeriod(TPM_Type *base, uint32_t ticks)

Sets the timer period in units of ticks.

Timers counts from 0 until it equals the count value set here. The count value is written to the MOD register.

Note:

a. This API allows the user to use the TPM module as a timer. Do not mix usage of this API with TPM's PWM setup API's.

b. Call the utility macros provided in the fsl_common.h to convert usec or msec to ticks.

Parameters

- base – TPM peripheral base address
- ticks – A timer period in units of ticks, which should be equal or greater than 1.

Returns
kStatus_Success PWM setup successful kStatus_Timeout PWM setup timeout when write register CnSC

static inline uint32_t TPM_GetCurrentTimerCount(TPM_Type *base)

Reads the current timer counting value.

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note: Call the utility macros provided in the fsl_common.h to convert ticks to usec or msec.

Parameters

- base – TPM peripheral base address

Returns
The current counter value in ticks

static inline void TPM_StartTimer(TPM_Type *base, *tpm_clock_source_t* clockSource)

Starts the TPM counter.

Parameters

- base – TPM peripheral base address
- clockSource – TPM clock source; once clock source is set the counter will start running

static inline *status_t* TPM_StopTimer(TPM_Type *base)

Stops the TPM counter.

Parameters

- base – TPM peripheral base address

Returns
kStatus_Success PWM setup successful kStatus_Timeout PWM setup timeout when write register CnSC

FSL_TPM_DRIVER_VERSION

TPM driver version 2.4.0.

enum _tpm_chnl

List of TPM channels.

Note: Actual number of available channels is SoC dependent

Values:

enumerator kTPM_Chnl_0
TPM channel number 0

enumerator kTPM_Chnl_1
TPM channel number 1

enumerator kTPM_Chnl_2
TPM channel number 2

enumerator kTPM_Chnl_3
TPM channel number 3

enumerator kTPM_Chnl_4
TPM channel number 4

enumerator kTPM_Chnl_5
TPM channel number 5

enumerator kTPM_Chnl_6
TPM channel number 6

enumerator kTPM_Chnl_7
TPM channel number 7

enum _tpm_pwm_mode
TPM PWM operation modes.

Values:

enumerator kTPM_EdgeAlignedPwm
Edge aligned PWM

enumerator kTPM_CenterAlignedPwm
Center aligned PWM

enumerator kTPM_CombinedPwm
Combined PWM (Edge-aligned, center-aligned, or asymmetrical PWMs can be obtained in combined mode using different software configurations)

enum _tpm_pwm_level_select
TPM PWM output pulse mode: high-true, low-true or no output.

Note: When the TPM has PWM pause level select feature, the PWM output cannot be turned off by selecting the output level. In this case, the channel must be closed to close the PWM output.

Values:

enumerator kTPM_HighTrue
High true pulses

enumerator kTPM_LowTrue
Low true pulses

enum _tpm_pwm_pause_level_select
TPM PWM output when first enabled or paused: set or clear.

Values:

enumerator kTPM_ClearOnPause
Clear Output when counter first enabled or paused.

enumerator kTPM_SetOnPause
Set Output when counter first enabled or paused.

enum _tpm_chnl_control_bit_mask
List of TPM channel modes and level control bit mask.

Values:

enumerator kTPM_ChnlELSnAMask
Channel ELSA bit mask.

enumerator kTPM_ChnlELSnBMask
Channel ELSB bit mask.

enumerator kTPM_ChnlMSAMask
Channel MSA bit mask.

enumerator kTPM_ChnlMSBMask
Channel MSB bit mask.

enum _tpm_trigger_select
Trigger sources available.

This is used for both internal & external trigger sources (external trigger sources available in certain SoC's)

Note: The actual trigger sources available is SoC-specific.

Values:

enumerator kTPM_Trigger_Select_0

enumerator kTPM_Trigger_Select_1

enumerator kTPM_Trigger_Select_2

enumerator kTPM_Trigger_Select_3

enumerator kTPM_Trigger_Select_4

enumerator kTPM_Trigger_Select_5

enumerator kTPM_Trigger_Select_6

enumerator kTPM_Trigger_Select_7

enumerator kTPM_Trigger_Select_8

enumerator kTPM_Trigger_Select_9

enumerator kTPM_Trigger_Select_10

enumerator kTPM_Trigger_Select_11

enumerator kTPM_Trigger_Select_12

enumerator kTPM_Trigger_Select_13

enumerator kTPM_Trigger_Select_14

enumerator kTPM_Trigger_Select_15

enum _tpm_trigger_source
Trigger source options available.

Note: This selection is available only on some SoC's. For SoC's without this selection, the only trigger source available is internal triger.

Values:

enumerator kTPM_TriggerSource_External
Use external trigger input

enumerator kTPM_TriggerSource_Internal
Use internal trigger (channel pin input capture)

enum _tpm_ext_trigger_polarity
External trigger source polarity.

Note: Selects the polarity of the external trigger source.

Values:

enumerator kTPM_ExtTrigger_Active_High
External trigger input is active high

enumerator kTPM_ExtTrigger_Active_Low
External trigger input is active low

enum _tpm_output_compare_mode
TPM output compare modes.

Values:

enumerator kTPM_NoOutputSignal
No channel output when counter reaches CnV

enumerator kTPM_ToggleOnMatch
Toggle output

enumerator kTPM_ClearOnMatch
Clear output

enumerator kTPM_SetOnMatch
Set output

enumerator kTPM_HighPulseOutput
Pulse output high

enumerator kTPM_LowPulseOutput
Pulse output low

enum _tpm_input_capture_edge
TPM input capture edge.

Values:

enumerator kTPM_RisingEdge
Capture on rising edge only

enumerator kTPM_FallingEdge
Capture on falling edge only

enumerator kTPM_RiseAndFallEdge
Capture on rising or falling edge

enum _tpm_quad_decode_mode
TPM quadrature decode modes.

Note: This mode is available only on some SoC's.

Values:

enumerator kTPM_QuadPhaseEncode
Phase A and Phase B encoding mode

enumerator kTPM_QuadCountAndDir
Count and direction encoding mode

enum _tpm_phase_polarity
TPM quadrature phase polarities.

Values:

enumerator kTPM_QuadPhaseNormal
Phase input signal is not inverted

enumerator kTPM_QuadPhaseInvert
Phase input signal is inverted

enum _tpm_clock_source
TPM clock source selection.

Values:

enumerator kTPM_SystemClock
System clock

enumerator kTPM_ExternalClock
External TPM_EXTCLK pin clock

enumerator kTPM_ExternalInputTriggerClock
Selected external input trigger clock

enum _tpm_clock_prescale
TPM prescale value selection for the clock source.

Values:

enumerator kTPM_Prescale_Divide_1
Divide by 1

enumerator kTPM_Prescale_Divide_2
Divide by 2

enumerator kTPM_Prescale_Divide_4
Divide by 4

enumerator kTPM_Prescale_Divide_8
Divide by 8

enumerator kTPM_Prescale_Divide_16
Divide by 16

enumerator kTPM_Prescale_Divide_32
Divide by 32

enumerator kTPM_Prescale_Divide_64
Divide by 64

enumerator kTPM_Prescale_Divide_128
Divide by 128

enum _tpm_interrupt_enable
List of TPM interrupts.

Values:

enumerator kTPM_Chnl0InterruptEnable
Channel 0 interrupt.

enumerator kTPM_Chnl1InterruptEnable
Channel 1 interrupt.

enumerator kTPM_Chnl2InterruptEnable
Channel 2 interrupt.

enumerator kTPM_Chnl3InterruptEnable
Channel 3 interrupt.

enumerator kTPM_Chnl4InterruptEnable
Channel 4 interrupt.

enumerator kTPM_Chnl5InterruptEnable
Channel 5 interrupt.

enumerator kTPM_Chnl6InterruptEnable
Channel 6 interrupt.

enumerator kTPM_Chnl7InterruptEnable
Channel 7 interrupt.

enumerator kTPM_TimeOverflowInterruptEnable
Time overflow interrupt.

enum _tpm_status_flags
List of TPM flags.

Values:

enumerator kTPM_Chnl0Flag
Channel 0 flag

enumerator kTPM_Chnl1Flag
Channel 1 flag

enumerator kTPM_Chnl2Flag
Channel 2 flag

enumerator kTPM_Chnl3Flag
Channel 3 flag

enumerator kTPM_Chnl4Flag
Channel 4 flag

enumerator kTPM_Chnl5Flag
Channel 5 flag

enumerator kTPM_Chnl6Flag
Channel 6 flag

enumerator kTPM_Chnl7Flag
Channel 7 flag

enumerator kTPM_TimeOverflowFlag
Time overflow flag

typedef enum *_tpm_chnl* tpm_chnl_t
List of TPM channels.

Note: Actual number of available channels is SoC dependent

typedef enum *_tpm_pwm_mode* tpm_pwm_mode_t

TPM PWM operation modes.

typedef enum *_tpm_pwm_level_select* tpm_pwm_level_select_t

TPM PWM output pulse mode: high-true, low-true or no output.

Note: When the TPM has PWM pause level select feature, the PWM output cannot be turned off by selecting the output level. In this case, the channel must be closed to close the PWM output.

typedef enum *_tpm_pwm_pause_level_select* tpm_pwm_pause_level_select_t

TPM PWM output when first enabled or paused: set or clear.

typedef enum *_tpm_chnl_control_bit_mask* tpm_chnl_control_bit_mask_t

List of TPM channel modes and level control bit mask.

typedef struct *_tpm_chnl_pwm_signal_param* tpm_chnl_pwm_signal_param_t

Options to configure a TPM channel's PWM signal.

typedef enum *_tpm_trigger_select* tpm_trigger_select_t

Trigger sources available.

This is used for both internal & external trigger sources (external trigger sources available in certain SoC's)

Note: The actual trigger sources available is SoC-specific.

typedef enum *_tpm_trigger_source* tpm_trigger_source_t

Trigger source options available.

Note: This selection is available only on some SoC's. For SoC's without this selection, the only trigger source available is internal triger.

typedef enum *_tpm_ext_trigger_polarity* tpm_ext_trigger_polarity_t

External trigger source polarity.

Note: Selects the polarity of the external trigger source.

typedef enum *_tpm_output_compare_mode* tpm_output_compare_mode_t

TPM output compare modes.

typedef enum *_tpm_input_capture_edge* tpm_input_capture_edge_t

TPM input capture edge.

typedef struct *_tpm_dual_edge_capture_param* tpm_dual_edge_capture_param_t

TPM dual edge capture parameters.

Note: This mode is available only on some SoC's.

typedef enum *_tpm_quad_decode_mode* tpm_quad_decode_mode_t

TPM quadrature decode modes.

Note: This mode is available only on some SoC's.

typedef enum *_tpm_phase_polarity* tpm_phase_polarity_t

TPM quadrature phase polarities.

typedef struct *_tpm_phase_param* tpm_phase_params_t

TPM quadrature decode phase parameters.

typedef enum *_tpm_clock_source* tpm_clock_source_t

TPM clock source selection.

typedef enum *_tpm_clock_prescale* tpm_clock_prescale_t

TPM prescale value selection for the clock source.

typedef struct *_tpm_config* tpm_config_t

TPM config structure.

This structure holds the configuration settings for the TPM peripheral. To initialize this structure to reasonable defaults, call the TPM_GetDefaultConfig() function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

typedef enum *_tpm_interrupt_enable* tpm_interrupt_enable_t

List of TPM interrupts.

typedef enum *_tpm_status_flags* tpm_status_flags_t

List of TPM flags.

typedef void (*tpm_callback_t)(TPM_Type *base)

TPM callback function pointer.

Param base
TPM peripheral base address.

TPM_TIMEOUT

Max loops to wait for writing register.

When writing MOD CnV CnSC and SC register, driver will wait until register is updated. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

TPM_MAX_COUNTER_VALUE(x)

Help macro to get the max counter value.

struct _tpm_chnl_pwm_signal_param

#include <fsl_tpm.h> Options to configure a TPM channel's PWM signal.

Public Members

tpm_chnl_t chnlNumber

TPM channel to configure. In combined mode (available in some SoC's), this represents the channel pair number

tpm_pwm_pause_level_select_t pauseLevel

PWM output level when counter first enabled or paused

tpm_pwm_level_select_t level

PWM output active level select

uint8_t dutyCyclePercent

PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=always active signal (100% duty cycle)

uint8_t firstEdgeDelayPercent

Used only in combined PWM mode to generate asymmetrical PWM. Specifies the delay to the first edge in a PWM period. If unsure, leave as 0. Should be specified as percentage of the PWM period, (dutyCyclePercent + firstEdgeDelayPercent) value should be not greate than 100.

bool enableComplementary

Used only in combined PWM mode. true: The combined channels output complementary signals; false: The combined channels output same signals;

tpm_pwm_pause_level_select_t secPauseLevel

Used only in combined PWM mode. Define the second channel output level when counter first enabled or paused

uint8_t deadTimeValue[2]

The dead time value for channel n and n+1 in combined complementary PWM mode. Deadtime insertion is disabled when this value is zero, otherwise deadtime insertion for channel n/n+1 is configured as (deadTimeValue * 4) clock cycles. deadTimeValue's available range is 0 ~ 15.

struct __tpm__dual__edge__capture__param

#include <fsl_tpm.h> TPM dual edge capture parameters.

Note: This mode is available only on some SoC's.

Public Members

bool enableSwap

true: Use channel n+1 input, channel n input is ignored; false: Use channel n input, channel n+1 input is ignored

tpm_input_capture_edge_t currChanEdgeMode

Input capture edge select for channel n

tpm_input_capture_edge_t nextChanEdgeMode

Input capture edge select for channel n+1

struct __tpm__phase__param

#include <fsl_tpm.h> TPM quadrature decode phase parameters.

Public Members

uint32_t phaseFilterVal

Filter value, filter is disabled when the value is zero

tpm_phase_polarity_t phasePolarity

Phase polarity

struct __tpm__config

#include <fsl_tpm.h> TPM config structure.

This structure holds the configuration settings for the TPM peripheral. To initialize this structure to reasonable defaults, call the TPM_GetDefaultConfig() function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

Public Members

tpm_clock_prescale_t prescale
Select TPM clock prescale value

bool useGlobalTimeBase
true: The TPM channels use an external global time base (the local counter still use for generate overflow interrupt and DMA request); false: All TPM channels use the local counter as their timebase

bool syncGlobalTimeBase
true: The TPM counter is synchronized to the global time base; false: disabled

tpm_trigger_select_t triggerSelect
Input trigger to use for controlling the counter operation

tpm_trigger_source_t triggerSource
Decides if we use external or internal trigger.

tpm_ext_trigger_polarity_t extTriggerPolarity
when using external trigger source, need selects the polarity of it.

bool enableDoze
true: TPM counter is paused in doze mode; false: TPM counter continues in doze mode

bool enableDebugMode
true: TPM counter continues in debug mode; false: TPM counter is paused in debug mode

bool enableReloadOnTrigger
true: TPM counter is reloaded on trigger; false: TPM counter not reloaded

bool enableStopOnOverflow
true: TPM counter stops after overflow; false: TPM counter continues running after overflow

bool enableStartOnTrigger
true: TPM counter only starts when a trigger is detected; false: TPM counter starts immediately

bool enablePauseOnTrigger
true: TPM counter will pause while trigger remains asserted; false: TPM counter continues running

uint8_t chnlPolarity
Defines the input/output polarity of the channels in POL register

2.63 TRDC: Trusted Resource Domain Controller

void TRDC_Init(TRDC_Type *base)
Initializes the TRDC module.

This function enables the TRDC clock.

Parameters

- base – TRDC peripheral base address.

void TRDC_Deinit(TRDC_Type *base)

De-initializes the TRDC module.

This function disables the TRDC clock.

Parameters

- base – TRDC peripheral base address.

static inline uint8_t TRDC_GetCurrentMasterDomainId(TRDC_Type *base)

Gets the domain ID of the current bus master.

Parameters

- base – TRDC peripheral base address.

Returns

Domain ID of current bus master.

void TRDC_GetHardwareConfig(TRDC_Type *base, *trdc_hardware_config_t* *config)

Gets the TRDC hardware configuration.

This function gets the TRDC hardware configurations, including number of bus masters, number of domains, number of MRCs and number of PACs.

Parameters

- base – TRDC peripheral base address.
- config – Pointer to the structure to get the configuration.

static inline void TRDC_SetDacGlobalValid(TRDC_Type *base)

Sets the TRDC DAC(Domain Assignment Controllers) global valid.

Once enabled, it will remain enabled until next reset.

Parameters

- base – TRDC peripheral base address.

static inline void TRDC_LockMasterDomainAssignment(TRDC_Type *base, uint8_t master)

Locks the bus master domain assignment register.

This function locks the master domain assignment. After it is locked, the register can't be changed until next reset.

Parameters

- base – TRDC peripheral base address.
- master – Which master to configure.

static inline void TRDC_SetMasterDomainAssignmentValid(TRDC_Type *base, uint8_t master, bool valid)

Sets the master domain assignment as valid or invalid.

This function sets the master domain assignment as valid or invalid.

Parameters

- base – TRDC peripheral base address.
- master – Which master to configure.
- valid – True to set valid, false to set invalid.

void TRDC_GetDefaultProcessorDomainAssignment(*trdc_processor_domain_assignment_t* *domainAssignment)

Gets the default master domain assignment for the processor bus master.

This function gets the default master domain assignment for the processor bus master. It should only be used for the processor bus masters, such as CORE0. This function sets the assignment as follows:

```
assignment->domainId          = 0U;
assignment->domainIdSelect    = kTRDC_DidMda;
assignment->lock              = 0U;
```

Parameters

- domainAssignment – Pointer to the assignment structure.

void TRDC_GetDefaultNonProcessorDomainAssignment(*trdc_non_processor_domain_assignment_t* *domainAssignment)

Gets the default master domain assignment for non-processor bus master.

This function gets the default master domain assignment for non-processor bus master. It should only be used for the non-processor bus masters, such as DMA. This function sets the assignment as follows:

```
assignment->domainId          = 0U;
assignment->privilegeAttr     = kTRDC_ForceUser;
assignment->secureAttr        = kTRDC_ForceSecure;
assignment->bypassDomainId    = 0U;
assignment->lock              = 0U;
```

Parameters

- domainAssignment – Pointer to the assignment structure.

void TRDC_SetProcessorDomainAssignment(TRDC_Type *base, const *trdc_processor_domain_assignment_t* *domainAssignment)

Sets the processor bus master domain assignment.

This function sets the processor master domain assignment as valid. One bus master might have multiple domain assignment registers. The parameter assignIndex specifies which assignment register to set.

Example: Set domain assignment for core 0.

```
trdc_processor_domain_assignment_t processorAssignment;

TRDC_GetDefaultProcessorDomainAssignment(&processorAssignment);

processorAssignment.domainId = 0;
processorAssignment.xxx      = xxx;
TRDC_SetMasterDomainAssignment(TRDC, &processorAssignment);
```

Parameters

- base – TRDC peripheral base address.
- domainAssignment – Pointer to the assignment structure.

static inline void TRDC_EnableProcessorDomainAssignment(TRDC_Type *base, bool enable)

Enables the processor bus master domain assignment.

Parameters

- base – TRDC peripheral base address.
- enable – True to enable, false to disable.

void TRDC_SetNonProcessorDomainAssignment(TRDC_Type *base, uint8_t master, const *trdc_non_processor_domain_assignment_t* *domainAssignment)

Sets the non-processor bus master domain assignment.

This function sets the non-processor master domain assignment as valid. One bus master might have multiple domain assignment registers. The parameter assignIndex specifies which assignment register to set.

Example: Set domain assignment for DMA0.

```
trdc_non_processor_domain_assignment_t nonProcessorAssignment;

TRDC_GetDefaultNonProcessorDomainAssignment(&nonProcessorAssignment);
nonProcessorAssignment.domainId = 1;
nonProcessorAssignment.xxx      = xxx;

TRDC_SetMasterDomainAssignment(TRDC, kTrdcMasterDma0, 0U, &nonProcessorAssignment);
```

Parameters

- base – TRDC peripheral base address.
- master – Which master to configure, refer to trdc_master_t in processor header file.
- domainAssignment – Pointer to the assignment structure.

void TRDC_GetDefaultIDAUConfig(*trdc_idau_config_t* *idauConfiguration)

Gets the default IDAU(Implementation-Defined Attribution Unit) configuration.

```
config->lockSecureVTOR    = false;
config->lockNonsecureVTOR = false;
config->lockSecureMPU     = false;
config->lockNonsecureMPU  = false;
config->lockSAU           = false;
```

Parameters

- idauConfiguration – Pointer to the configuration structure.

void TRDC_SetIDAU(TRDC_Type *base, const *trdc_idau_config_t* *idauConfiguration)

Sets the IDAU(Implementation-Defined Attribution Unit) control configuration.

Example: Lock the secure and non-secure MPU registers.

```
trdc_idau_config_t idauConfiguration;

TRDC_GetDefaultIDAUConfig(&idauConfiguration);

idauConfiguration.lockSecureMPU = true;
idauConfiguration.lockNonsecureMPU      = true;
TRDC_SetIDAU(TRDC, &idauConfiguration);
```

Parameters

- base – TRDC peripheral base address.
- idauConfiguration – Pointer to the configuration structure.

static inline void TRDC_EnableFlashLogicalWindow(TRDC_Type *base, bool enable)

Enables/disables the FLW(flash logical window) function.

Parameters

- base – TRDC peripheral base address.
- enable – True to enable, false to disable.

static inline void TRDC_LockFlashLogicalWindow(TRDC_Type *base)

Locks FLW registers. Once locked the registers can noy be updated until next reset.

Parameters

- base – TRDC peripheral base address.

static inline uint32_t TRDC_GetFlashLogicalWindowPbase(TRDC_Type *base)

Gets the FLW physical base address.

Parameters

- base – TRDC peripheral base address.

Returns

Physical address of the FLW function.

static inline void TRDC_GetSetFlashLogicalWindowSize(TRDC_Type *base, uint16_t size)

Sets the FLW size.

Parameters

- base – TRDC peripheral base address.
- size – Size of the FLW in unit of 32k bytes.

void TRDC_GetDefaultFlashLogicalWindowConfig(*trdc_flw_config_t* *flwConfiguration)

Gets the default FLW(Flsh Logical Window) configuration.

```
config->blockCount     = false;
config->arrayBaseAddr = false;
config->lock      = false;
config->enable   = false;
```

Parameters

- flwConfiguration – Pointer to the configuration structure.

void TRDC_SetFlashLogicalWindow(TRDC_Type *base, const *trdc_flw_config_t* *flwConfiguration)

Sets the FLW function's configuration.

```
trdc_flw_config_t flwConfiguration;

TRDC_GetDefaultIDAUConfig(&flwConfiguration);

flwConfiguration.blockCount = 32U;
flwConfiguration.arrayBaseAddr = 0xXXXXXXXX;
TRDC_SetIDAU(TRDC, &flwConfiguration);
```

Parameters

- base – TRDC peripheral base address.
- flwConfiguration – Pointer to the configuration structure.

status_t TRDC_GetAndClearFirstDomainError(TRDC_Type *base, *trdc_domain_error_t* *error)

Gets and clears the first domain error of the current domain.

This function gets the first access violation information for the current domain and clears the pending flag. There might be multiple access violations pending for the current domain. This function only processes the first error.

Parameters

- base – TRDC peripheral base address.
- error – Pointer to the error information.

Returns

If the access violation is captured, this function returns the kStatus_Success. The error information can be obtained from the parameter error. If no access violation is captured, this function returns the kStatus_NoData.

status_t TRDC_GetAndClearFirstSpecificDomainError(TRDC_Type *base, *trdc_domain_error_t* *error, uint8_t domainId)

Gets and clears the first domain error of the specific domain.

This function gets the first access violation information for the specific domain and clears the pending flag. There might be multiple access violations pending for the current domain. This function only processes the first error.

Parameters

- base – TRDC peripheral base address.
- error – Pointer to the error information.
- domainId – The error of which domain to get and clear.

Returns

If the access violation is captured, this function returns the kStatus_Success. The error information can be obtained from the parameter error. If no access violation is captured, this function returns the kStatus_NoData.

static inline void TRDC_SetMrcGlobalValid(TRDC_Type *base)

Sets the TRDC MRC(Memory Region Checkers) global valid.

Once enabled, it will remain enabled until next reset.

Parameters

- base – TRDC peripheral base address.

static inline uint8_t TRDC_GetMrcRegionNumber(TRDC_Type *base, uint8_t mrcIdx)

Gets the TRDC MRC(Memory Region Checkers) region number valid.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.

Returns

the region number of the given MRC instance

void TRDC_MrcSetMemoryAccessConfig(TRDC_Type *base, const *trdc_memory_access_control_config_t* *config, uint8_t mrcIdx, uint8_t regIdx)

Sets the memory access configuration for one of the access control register of one MRC.

Example: Enable the secure operations and lock the configuration for MRC0 region 1.

```
trdc_memory_access_control_config_t config;

config.securePrivX = true;
config.securePrivW = true;
config.securePrivR = true;
config.lock = true;
TRDC_SetMrcMemoryAccess(TRDC, &config, 0, 1);
```

Parameters

- base – TRDC peripheral base address.
- config – Pointer to the configuration structure.
- mrcIdx – MRC index.
- regIdx – Register number.

void TRDC_MrcEnableDomainNseUpdate(TRDC_Type *base, uint8_t mrcIdx, uint16_t domainMask, bool enable)

Enables the update of the selected domains.

After the domains' update are enabled, their regions' NSE bits can be set or clear.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.
- domainMask – Bit mask of the domains to be enabled.
- enable – True to enable, false to disable.

void TRDC_MrcRegionNseSet(TRDC_Type *base, uint8_t mrcIdx, uint16_t regionMask)

Sets the NSE bits of the selected regions for domains.

This function sets the NSE bits for the selected regions for the domains whose update are enabled.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.
- regionMask – Bit mask of the regions whose NSE bits to set.

void TRDC_MrcRegionNseClear(TRDC_Type *base, uint8_t mrcIdx, uint16_t regionMask)

Clears the NSE bits of the selected regions for domains.

This function clears the NSE bits for the selected regions for the domains whose update are enabled.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.
- regionMask – Bit mask of the regions whose NSE bits to clear.

void TRDC_MrcDomainNseClear(TRDC_Type *base, uint8_t mrcIdx, uint16_t domainMask)

Clears the NSE bits for all the regions of the selected domains.

This function clears the NSE bits for all regions of selected domains whose update are enabled.

Parameters

- base – TRDC peripheral base address.
- mrcIdx – MRC index.
- domainMask – Bit mask of the domains whose NSE bits to clear.

void TRDC_MrcSetRegionDescriptorConfig(TRDC_Type *base, const *trdc_mrc_region_descriptor_config_t* *config)

Sets the configuration for one of the region descriptor per domain per MRC instnce.

This function sets the configuration for one of the region descriptor, including the start and end address of the region, memory access control policy and valid.

Parameters

- base – TRDC peripheral base address.
- config – Pointer to region descriptor configuration structure.

static inline void TRDC_SetMbcGlobalValid(TRDC_Type *base)

Sets the TRDC MBC(Memory Block Checkers) global valid.

Once enabled, it will remain enabled until next reset.

Parameters

- base – TRDC peripheral base address.

void TRDC_GetMbcHardwareConfig(TRDC_Type *base, *trdc_slave_memory_hardware_config_t* *config, uint8_t mbcIdx, uint8_t slvIdx)

Gets the hardware configuration of the one of two slave memories within each MBC(memory block checker).

Parameters

- base – TRDC peripheral base address.
- config – Pointer to the structure to get the configuration.
- mbcIdx – MBC number.
- slvIdx – Slave number.

void TRDC_MbcSetNseUpdateConfig(TRDC_Type *base, const *trdc_mbc_nse_update_config_t* *config, uint8_t mbcIdx)

Sets the NSR update configuration for one of the MBC instance.

After set the NSE configuration, the configured memory area can be updateby NSE set/clear.

Parameters

- base – TRDC peripheral base address.
- config – Pointer to NSE update configuration structure.
- mbcIdx – MBC index.

void TRDC_MbcWordNseSet(TRDC_Type *base, uint8_t mbcIdx, uint32_t bitMask)

Sets the NSE bits of the selected configuration words according to NSE update configuration.

This function sets the NSE bits of the word for the configured regio, memory.

Parameters

- base – TRDC peripheral base address.
- mbcIdx – MBC index.
- bitMask – Mask of the bits whose NSE bits to set.

void TRDC_MbcWordNseClear(TRDC_Type *base, uint8_t mbcIdx, uint32_t bitMask)

Clears the NSE bits of the selected configuration words according to NSE update configuration.

This function sets the NSE bits of the word for the configured regio, memory.

Parameters

- base – TRDC peripheral base address.
- mbcIdx – MBC index.
- bitMask – Mask of the bits whose NSE bits to clear.

void TRDC_MbcNseClearAll(TRDC_Type *base, uint8_t mbcIdx, uint16_t domainMask, uint8_t slaveMask)

Clears all configuration words' NSE bits of the selected domain and memory.

Parameters

- base – TRDC peripheral base address.
- mbcIdx – MBC index.
- domainMask – Mask of the domains whose NSE bits to clear, 0b110 means clear domain 1&2.
- slaveMask – Mask of the slaves whose NSE bits to clear, 0x11 means clear all slave 0&1's NSE bits.

void TRDC_MbcSetMemoryAccessConfig(TRDC_Type *base, const *trdc_memory_access_control_config_t* *config, uint8_t mbcIdx, uint8_t rgdIdx)

Sets the memory access configuration for one of the region descriptor of one MBC.

Example: Enable the secure operations and lock the configuration for MRC0 region 1.

```
trdc_memory_access_control_config_t config;

config.securePrivX = true;
config.securePrivW = true;
config.securePrivR = true;
config.lock = true;
TRDC_SetMbcMemoryAccess(TRDC, &config, 0, 1);
```

Parameters

- base – TRDC peripheral base address.
- config – Pointer to the configuration structure.
- mbcIdx – MBC index.
- rgdIdx – Region descriptor number.

void TRDC_MbcSetMemoryBlockConfig(TRDC_Type *base, const *trdc_mbc_memory_block_config_t* *config)

Sets the configuration for one of the memory block per domain per MBC instnce.

This function sets the configuration for one of the memory block, including the memory access control policy and nse enable.

Parameters

- base – TRDC peripheral base address.
- config – Pointer to memory block configuration structure.

enum _trdc_did_sel

TRDC domain ID select method, the register bit TRDC_MDA_W0_0_DFMT0[DIDS], used for domain hit evaluation.

Values:

enumerator kTRDC_DidMda

Use MDAn[2:0] as DID.

enumerator kTRDC_DidInput

Use the input DID (DID_in) as DID.

enumerator kTRDC_DidMdaAndInput

Use MDAn[2] concatenated with DID_in[1:0] as DID.

enumerator kTRDC_DidReserved

Reserved.

enum _trdc_secure_attr

TRDC secure attribute, the register bit TRDC_MDA_W0_0_DFMT0[SA], used for bus master domain assignment.

Values:

enumerator kTRDC_ForceSecure

Force the bus attribute for this master to secure.

enumerator kTRDC_ForceNonSecure

Force the bus attribute for this master to non-secure.

enumerator kTRDC_MasterSecure

Use the bus master's secure/nonsecure attribute directly.

enumerator kTRDC_MasterSecure1

Use the bus master's secure/nonsecure attribute directly.

enum _trdc_privilege_attr

TRDC privileged attribute, the register bit TRDC_MDA_W0_x_DFMT1[PA], used for non-processor bus master domain assignment.

Values:

enumerator kTRDC_ForceUser

Force the bus attribute for this master to user.

enumerator kTRDC_ForcePrivilege

Force the bus attribute for this master to privileged.

enumerator kTRDC_MasterPrivilege

Use the bus master's attribute directly.

enumerator kTRDC_MasterPrivilege1

Use the bus master's attribute directly.

enum _trdc_controller

TRDC controller definition for domain error check. Each TRDC instance may have different MRC or MBC count, call TRDC_GetHardwareConfig to get the actual count.

Values:

enumerator kTRDC_MemBlockController0

Memory block checker 0.

enumerator kTRDC_MemBlockController1
Memory block checker 1.

enumerator kTRDC_MemBlockController2
Memory block checker 2.

enumerator kTRDC_MemBlockController3
Memory block checker 3.

enumerator kTRDC_MemRegionChecker0
Memory region checker 0.

enumerator kTRDC_MemRegionChecker1
Memory region checker 1.

enumerator kTRDC_MemRegionChecker2
Memory region checker 2.

enumerator kTRDC_MemRegionChecker3
Memory region checker 3.

enumerator kTRDC_MemRegionChecker4
Memory region checker 4.

enumerator kTRDC_MemRegionChecker5
Memory region checker 5.

enumerator kTRDC_MemRegionChecker6
Memory region checker 6.

enum __trdc_error_state
TRDC domain error state definition TRDC_MBCn_DERR_W1[EST] or TRDC_MRCn_DERR_W1[EST].

Values:

enumerator kTRDC_ErrorStateNone
No access violation detected.

enumerator kTRDC_ErrorStateNone1
No access violation detected.

enumerator kTRDC_ErrorStateSingle
Single access violation detected.

enumerator kTRDC_ErrorStateMulti
Multiple access violation detected.

enum __trdc_error_attr
TRDC domain error attribute definition TRDC_MBCn_DERR_W1[EATR] or TRDC_MRCn_DERR_W1[EATR].

Values:

enumerator kTRDC_ErrorSecureUserInst
Secure user mode, instruction fetch access.

enumerator kTRDC_ErrorSecureUserData
Secure user mode, data access.

enumerator kTRDC_ErrorSecurePrivilegeInst
Secure privileged mode, instruction fetch access.

enumerator kTRDC_ErrorSecurePrivilegeData
Secure privileged mode, data access.

enumerator kTRDC_ErrorNonSecureUserInst
NonSecure user mode, instruction fetch access.

enumerator kTRDC_ErrorNonSecureUserData
NonSecure user mode, data access.

enumerator kTRDC_ErrorNonSecurePrivilegeInst
NonSecure privileged mode, instruction fetch access.

enumerator kTRDC_ErrorNonSecurePrivilegeData
NonSecure privileged mode, data access.

enum _trdc_error_type
TRDC domain error access type definition TRDC_DERR_W1_n[ERW].

Values:

enumerator kTRDC_ErrorTypeRead
Error occurs on read reference.

enumerator kTRDC_ErrorTypeWrite
Error occurs on write reference.

enum _trdc_region_descriptor
The region descriptor enumeration, used to form a mask to set/clear the NSE bits for one or several regions.

Values:

enumerator kTRDC_RegionDescriptor0
Region descriptor 0.

enumerator kTRDC_RegionDescriptor1
Region descriptor 1.

enumerator kTRDC_RegionDescriptor2
Region descriptor 2.

enumerator kTRDC_RegionDescriptor3
Region descriptor 3.

enumerator kTRDC_RegionDescriptor4
Region descriptor 4.

enumerator kTRDC_RegionDescriptor5
Region descriptor 5.

enumerator kTRDC_RegionDescriptor6
Region descriptor 6.

enumerator kTRDC_RegionDescriptor7
Region descriptor 7.

enumerator kTRDC_RegionDescriptor8
Region descriptor 8.

enumerator kTRDC_RegionDescriptor9
Region descriptor 9.

enumerator kTRDC_RegionDescriptor10
Region descriptor 10.

enumerator kTRDC_RegionDescriptor11
Region descriptor 11.

enumerator kTRDC_RegionDescriptor12
Region descriptor 12.

enumerator kTRDC_RegionDescriptor13
Region descriptor 13.

enumerator kTRDC_RegionDescriptor14
Region descriptor 14.

enumerator kTRDC_RegionDescriptor15
Region descriptor 15.

enum _trdc_MRC_domain
The MRC domain enumeration, used to form a mask to enable/disable the update or clear all NSE bits of one or several domains.

Values:

enumerator kTRDC_MrcDomain0
Domain 0.

enumerator kTRDC_MrcDomain1
Domain 1.

enumerator kTRDC_MrcDomain2
Domain 2.

enumerator kTRDC_MrcDomain3
Domain 3.

enumerator kTRDC_MrcDomain4
Domain 4.

enumerator kTRDC_MrcDomain5
Domain 5.

enumerator kTRDC_MrcDomain6
Domain 6.

enumerator kTRDC_MrcDomain7
Domain 7.

enumerator kTRDC_MrcDomain8
Domain 8.

enumerator kTRDC_MrcDomain9
Domain 9.

enumerator kTRDC_MrcDomain10
Domain 10.

enumerator kTRDC_MrcDomain11
Domain 11.

enumerator kTRDC_MrcDomain12
Domain 12.

enumerator kTRDC_MrcDomain13
Domain 13.

enumerator kTRDC_MrcDomain14
Domain 14.

enumerator kTRDC_MrcDomain15
Domain 15.

enum _trdc_MBC_domain
The MBC domain enumeration, used to form a mask to enable/disable the update or clear NSE bits of one or several domains.

Values:

enumerator kTRDC_MbcDomain0
Domain 0.

enumerator kTRDC_MbcDomain1
Domain 1.

enumerator kTRDC_MbcDomain2
Domain 2.

enumerator kTRDC_MbcDomain3
Domain 3.

enumerator kTRDC_MbcDomain4
Domain 4.

enumerator kTRDC_MbcDomain5
Domain 5.

enumerator kTRDC_MbcDomain6
Domain 6.

enumerator kTRDC_MbcDomain7
Domain 7.

enum _trdc_MBC_memory
The MBC slave memory enumeration, used to form a mask to enable/disable the update or clear NSE bits of one or several memory block.

Values:

enumerator kTRDC_MbcSlaveMemory0
Memory 0.

enumerator kTRDC_MbcSlaveMemory1
Memory 1.

enumerator kTRDC_MbcSlaveMemory2
Memory 2.

enumerator kTRDC_MbcSlaveMemory3
Memory 3.

enum _trdc_MBC_bit
The MBC bit enumeration, used to form a mask to set/clear configured words' NSE.

Values:

enumerator kTRDC_MbcBit0
Bit 0.

enumerator kTRDC_MbcBit1
Bit 1.

enumerator kTRDC_MbcBit2
Bit 2.

enumerator kTRDC_MbcBit3
Bit 3.

enumerator kTRDC_MbcBit4
Bit 4.

enumerator kTRDC_MbcBit5
Bit 5.

enumerator kTRDC_MbcBit6
Bit 6.

enumerator kTRDC_MbcBit7
Bit 7.

enumerator kTRDC_MbcBit8
Bit 8.

enumerator kTRDC_MbcBit9
Bit 9.

enumerator kTRDC_MbcBit10
Bit 10.

enumerator kTRDC_MbcBit11
Bit 11.

enumerator kTRDC_MbcBit12
Bit 12.

enumerator kTRDC_MbcBit13
Bit 13.

enumerator kTRDC_MbcBit14
Bit 14.

enumerator kTRDC_MbcBit15
Bit 15.

enumerator kTRDC_MbcBit16
Bit 16.

enumerator kTRDC_MbcBit17
Bit 17.

enumerator kTRDC_MbcBit18
Bit 18.

enumerator kTRDC_MbcBit19
Bit 19.

enumerator kTRDC_MbcBit20
Bit 20.

enumerator kTRDC_MbcBit21
Bit 21.

enumerator kTRDC_MbcBit22
Bit 22.

enumerator kTRDC_MbcBit23
Bit 23.

enumerator kTRDC_MbcBit24
Bit 24.

enumerator kTRDC_MbcBit25
Bit 25.

enumerator kTRDC_MbcBit26
Bit 26.

enumerator kTRDC_MbcBit27
Bit 27.

enumerator kTRDC_MbcBit28
Bit 28.

enumerator kTRDC_MbcBit29
Bit 29.

enumerator kTRDC_MbcBit30
Bit 30.

enumerator kTRDC_MbcBit31
Bit 31.

typedef struct *_trdc_hardware_config* trdc_hardware_config_t
TRDC hardware configuration.

typedef struct *_trdc_slave_memory_hardware_config* trdc_slave_memory_hardware_config_t
Hardware configuration of the two slave memories within each MBC(memory block checker).

typedef enum *_trdc_did_sel* trdc_did_sel_t
TRDC domain ID select method, the register bit TRDC_MDA_W0_0_DFMT0[DIDS], used for domain hit evaluation.

typedef enum *_trdc_secure_attr* trdc_secure_attr_t
TRDC secure attribute, the register bit TRDC_MDA_W0_0_DFMT0[SA], used for bus master domain assignment.

typedef struct *_trdc_processor_domain_assignment* trdc_processor_domain_assignment_t
Domain assignment for the processor bus master.

typedef enum *_trdc_privilege_attr* trdc_privilege_attr_t
TRDC privileged attribute, the register bit TRDC_MDA_W0_x_DFMT1[PA], used for non-processor bus master domain assignment.

typedef struct *_trdc_non_processor_domain_assignment* trdc_non_processor_domain_assignment_t
Domain assignment for the non-processor bus master.

typedef struct *_trdc_idau_config* trdc_idau_config_t
IDAU(Implementation-Defined Attribution Unit) configuration for TZ-M function control.

typedef struct *_trdc_flw_config* trdc_flw_config_t
FLW(Flash Logical Window) configuration.

typedef enum *_trdc_controller* trdc_controller_t
TRDC controller definition for domain error check. Each TRDC instance may have different MRC or MBC count, call TRDC_GetHardwareConfig to get the actual count.

typedef enum *_trdc_error_state* trdc_error_state_t

TRDC domain error state definition TRDC_MBCn_DERR_W1[EST] or TRDC_MRCn_DERR_W1[EST].

typedef enum *_trdc_error_attr* trdc_error_attr_t

TRDC domain error attribute definition TRDC_MBCn_DERR_W1[EATR] or TRDC_MRCn_DERR_W1[EATR].

typedef enum *_trdc_error_type* trdc_error_type_t

TRDC domain error access type definition TRDC_DERR_W1_n[ERW].

typedef struct *_trdc_domain_error* trdc_domain_error_t

TRDC domain error definition.

typedef struct *_trdc_memory_access_control_config* trdc_memory_access_control_config_t

Memory access control configuration for MBC/MRC.

typedef struct *_trdc_mrc_region_descriptor_config* trdc_mrc_region_descriptor_config_t

The configuration of each region descriptor per domain per MRC instance.

typedef struct *_trdc_mbc_nse_update_config* trdc_mbc_nse_update_config_t

The configuration of MBC NSE update.

typedef struct *_trdc_mbc_memory_block_config* trdc_mbc_memory_block_config_t

The configuration of each memory block per domain per MBC instance.

FSL_TRDC_DRIVER_VERSION

struct _trdc_hardware_config

#include <fsl_trdc.h> TRDC hardware configuration.

Public Members

uint8_t masterNumber

Number of bus masters.

uint8_t domainNumber

Number of domains.

uint8_t mbcNumber

Number of MBCs.

uint8_t mrcNumber

Number of MRCs.

struct _trdc_slave_memory_hardware_config

#include <fsl_trdc.h> Hardware configuration of the two slave memories within each MBC(memory block checker).

Public Members

uint32_t blockNum

Number of blocks.

uint32_t blockSize

Block size.

struct _trdc_processor_domain_assignment

#include <fsl_trdc.h> Domain assignment for the processor bus master.

Public Members

uint32_t domainId
Domain ID.

uint32_t domainIdSelect
Domain ID select method, see trdc_did_sel_t.

uint32_t ___pad0___
Reserved.

uint32_t secureAttr
Secure attribute, see trdc_secure_attr_t.

uint32_t ___pad1___
Reserved.

uint32_t lock
Lock the register.

uint32_t ___pad2___
Reserved.

struct __trdc__non__processor__domain__assignment
#include <fsl_trdc.h> Domain assignment for the non-processor bus master.

Public Members

uint32_t domainId
Domain ID.

uint32_t privilegeAttr
Privileged attribute, see trdc_privilege_attr_t.

uint32_t secureAttr
Secure attribute, see trdc_secure_attr_t.

uint32_t bypassDomainId
Bypass domain ID.

uint32_t ___pad0___
Reserved.

uint32_t lock
Lock the register.

uint32_t ___pad1___
Reserved.

struct __trdc__idau__config
#include <fsl_trdc.h> IDAU(Implementation-Defined Attribution Unit) configuration for TZ-M function control.

Public Members

uint32_t ___pad0___
Reserved.

uint32_t lockSecureVTOR
Disable writes to secure VTOR(Vector Table Offset Register).

uint32_t lockNonsecureVTOR
Disable writes to non-secure VTOR, Application interrupt and Reset Control Registers.

uint32_t lockSecureMPU
Disable writes to secure MPU(Memory Protection Unit) from software or from a debug agent connected to the processor in Secure state.

uint32_t lockNonsecureMPU
Disable writes to non-secure MPU(Memory Protection Unit) from software or from a debug agent connected to the processor.

uint32_t lockSAU
Disable writes to SAU(Security Attribution Unit) registers.

uint32_t ___pad1___
Reserved.

struct __trdc__flw__config
#include <fsl_trdc.h> FLW(Flash Logical Window) configuration.

Public Members

uint16_t blockCount
Block count of the Flash Logic Window in 32KByte blocks.

uint32_t arrayBaseAddr
Flash array base address of the Flash Logical Window.

bool lock
Disable writes to FLW registers.

bool enable
Enable FLW function.

struct __trdc__domain__error
#include <fsl_trdc.h> TRDC domain error definition.

Public Members

trdc_controller_t controller
Which controller captured access violation.

uint32_t address
Access address that generated access violation.

trdc_error_state_t errorState
Error state.

trdc_error_attr_t errorAttr
Error attribute.

trdc_error_type_t errorType
Error type.

uint8_t errorPort
Error port.

uint8_t domainId
Domain ID.

uint8_t slaveMemoryIdx
The slave memory index. Only apply when violation in MBC.

struct __trdc__memory__access__control__config
#include <fsl_trdc.h> Memory access control configuration for MBC/MRC.

Public Members

uint32_t nonsecureUsrX
Allow nonsecure user execute access.

uint32_t nonsecureUsrW
Allow nonsecure user write access.

uint32_t nonsecureUsrR
Allow nonsecure user read access.

uint32_t ___pad0___
Reserved.

uint32_t nonsecurePrivX
Allow nonsecure privilege execute access.

uint32_t nonsecurePrivW
Allow nonsecure privilege write access.

uint32_t nonsecurePrivR
Allow nonsecure privilege read access.

uint32_t ___pad1___
Reserved.

uint32_t secureUsrX
Allow secure user execute access.

uint32_t secureUsrW
Allow secure user write access.

uint32_t secureUsrR
Allow secure user read access.

uint32_t ___pad2___
Reserved.

uint32_t securePrivX
Allownsecure privilege execute access.

uint32_t securePrivW
Allownsecure privilege write access.

uint32_t securePrivR
Allownsecure privilege read access.

uint32_t ___pad3___
Reserved.

uint32_t lock
Lock the configuration until next reset, only apply to access control register 0.

struct __trdc__mrc__region__descriptor__config
#include <fsl_trdc.h> The configuration of each region descriptor per domain per MRC instance.

Public Members

uint8_t memoryAccessControlSelect
: Select one of the 8 access control policies for this region, for access cotrol policies see trdc_memory_access_control_config_t.

uint32_t startAddr
: Physical start address.

bool valid
: Lock the register.

bool nseEnable
: Enable non-secure accesses and disable secure accesses.

uint32_t endAddr
: Physical start address.

uint8_t mrcIdx
: The index of the MRC for this configuration to take effect.

uint8_t domainIdx
: The index of the domain for this configuration to take effect.

uint8_t regionIdx
: The index of the region for this configuration to take effect.

struct __trdc__mbc__nse__update__config
: *#include <fsl_trdc.h>* The configuration of MBC NSE update.

Public Members

uint32_t ___pad0___
: Reserved.

uint32_t wordIdx
: MBC configuration word index to be updated.

uint32_t ___pad1___
: Reserved.

uint32_t memorySelect
: Bit mask of the selected memory to be updated. _trdc_MBC_memory.

uint32_t ___pad2___
: Reserved.

uint32_t domianSelect
: Bit mask of the selected domain to be updated. _trdc_MBC_domain.

uint32_t ___pad3___
: Reserved.

uint32_t autoIncrement
: Whether to increment the word index after current word is updated using this configuration.

struct __trdc__mbc__memory__block__config
: *#include <fsl_trdc.h>* The configuration of each memory block per domain per MBC instance.

Public Members

uint32_t memoryAccessControlSelect
: Select one of the 8 access control policies for this memory block, for access cotrol policies see trdc_memory_access_control_config_t.

uint32_t nseEnable
: Enable non-secure accesses and disable secure accesses.

uint32_t mbcIdx
: The index of the MBC for this configuration to take effect.

uint32_t domainIdx
: The index of the domain for this configuration to take effect.

uint32_t slaveMemoryIdx
: The index of the slave memory for this configuration to take effect.

uint32_t memoryBlockIdx
: The index of the memory block for this configuration to take effect.

2.64 TRGMUX: Trigger Mux Driver

static inline void TRGMUX_LockRegister(TRGMUX_Type *base, uint32_t index)

Sets the flag of the register which is used to mark writeable.

The function sets the flag of the register which is used to mark writeable. Example:

```
TRGMUX_LockRegister(TRGMUX0,kTRGMUX_Trgmux0Dmamux0);
```

Parameters

- base – TRGMUX peripheral base address.
- index – The index of the TRGMUX register, see the enum trgmux_device_t defined in <SOC>.h.

status_t TRGMUX_SetTriggerSource(TRGMUX_Type *base, uint32_t index, *trgmux_trigger_input_t* input, uint32_t trigger_src)

Configures the trigger source of the appointed peripheral.

The function configures the trigger source of the appointed peripheral. Example:

```
TRGMUX_SetTriggerSource(TRGMUX0, kTRGMUX_Trgmux0Dmamux0, kTRGMUX_TriggerInput0,
↪ kTRGMUX_SourcePortPin);
```

Parameters

- base – TRGMUX peripheral base address.
- index – The index of the TRGMUX register, see the enum trgmux_device_t defined in <SOC>.h.
- input – The MUX select for peripheral trigger input
- trigger_src – The trigger inputs for various peripherals. See the enum trgmux_source_t defined in <SOC>.h.

Return values

- kStatus_Success – Configured successfully.

- kStatus_TRGMUX_Locked – Configuration failed because the register is locked.

FSL_TRGMUX_DRIVER_VERSION

TRGMUX driver version.

TRGMUX configure status.

Values:

enumerator kStatus_TRGMUX_Locked

Configure failed for register is locked

enum _trgmux_trigger_input

Defines the MUX select for peripheral trigger input.

Values:

enumerator kTRGMUX_TriggerInput0

The MUX select for peripheral trigger input 0

enumerator kTRGMUX_TriggerInput1

The MUX select for peripheral trigger input 1

enumerator kTRGMUX_TriggerInput2

The MUX select for peripheral trigger input 2

enumerator kTRGMUX_TriggerInput3

The MUX select for peripheral trigger input 3

typedef enum *_trgmux_trigger_input* trgmux_trigger_input_t

Defines the MUX select for peripheral trigger input.

2.65 TSTMR: Timestamp Timer Driver

void TSTMR_Init(TSTMR_Type *base)

Init TSTMR.

This function initializes the TSTMR module.

Parameters

- base – TSTMR peripheral base address.

void TSTMR_Deinit(TSTMR_Type *base)

Deinit TSTMR.

This function deinitializes the TSTMR module.

Parameters

- base – TSTMR peripheral base address.

FSL_TSTMR_DRIVER_VERSION

Version 2.1.0

static inline uint64_t TSTMR_ReadTimeStamp(TSTMR_Type *base)

Reads the time stamp.

This function reads the low and high registers and returns the 56-bit free running counter value. This can be read by software at any time to determine the software ticks. TSTMR registers can be read with 32-bit accesses only. The TSTMR LOW read should occur first, followed by the TSTMR HIGH read.

Parameters

- base – TSTMR peripheral base address.

Returns
The 56-bit time stamp value.

void TSTMR_DelayUs(TSTMR_Type *base, uint64_t delayInUs)

Delays for a specified number of microseconds.

This function repeatedly reads the timestamp register and waits for the user-specified delay value.

Parameters

- base – TSTMR peripheral base address.
- delayInUs – Delay value in microseconds.

2.66 VBAT: Smart Power Switch

The enumeration of VBAT module status.

Values:

enumerator kStatus_VBAT_Fro16kNotEnabled
Internal 16kHz free running oscillator not enabled.

enumerator kStatus_VBAT_BandgapNotEnabled
Bandgap not enabled.

enum _vbat_status_flag

The enumeration of VBAT status flags.

Values:

enumerator kVBAT_StatusFlagPORDetect
VBAT domain has been reset

enumerator kVBAT_StatusFlagWakeupPin
A falling edge is detected on the wakeup pin.

enumerator kVBAT_StatusFlagBandgapTimer0
Bandgap Timer0 period reached.

enumerator kVBAT_StatusFlagBandgapTimer1
Bandgap Timer1 period reached.

enumerator kVBAT_StatusFlagLdoReady
LDO is enabled and ready.

enum _vbat_interrupt_enable

The enumeration of VBAT interrupt enable.

Values:

enumerator kVBAT_InterruptEnablePORDetect
Enable POR detect interrupt.

enumerator kVBAT_InterruptEnableWakeupPin
Enable the interrupt when a falling edge is detected on the wakeup pin.

enumerator kVBAT_InterruptEnableBandgapTimer0
Enable the interrupt if Bandgap Timer0 period reached.

enumerator kVBAT_InterruptEnableBandgapTimer1
Enable the interrupt if Bandgap Timer1 period reached.

enumerator kVBAT_InterruptEnableLdoReady
Enable LDO ready interrupt.

enumerator kVBAT_AllInterruptsEnable
Enable all interrupts.

enum _vbat_wakeup_enable
The enumeration of VBAT wakeup enable.

Values:

enumerator kVBAT_WakeupEnablePORDetect
Enable POR detect wakeup.

enumerator kVBAT_WakeupEnableWakeupPin
Enable wakeup feature when a falling edge is detected on the wakeup pin.

enumerator kVBAT_WakeupEnableBandgapTimer0
Enable wakeup feature when bandgap timer0 period reached.

enumerator kVBAT_WakeupEnableBandgapTimer1
Enable wakeup feature when bandgap timer1 period reached.

enumerator kVBAT_WakeupEnableLdoReady
Enable wakeup when LDO ready.

enumerator kVBAT_AllWakeupsEnable
Enable all wakeup.

enum _vbat_bandgap_timer_id
The enumeration of bandgap timer id, VBAT support two bandgap timers.

Values:

enumerator kVBAT_BandgapTimer0
Bandgap Timer0.

enumerator kVBAT_BandgapTimer1
Bandgap Timer1.

enum _vbat_bandgap_refresh_period
The enumeration of bandgap refresh period.

Values:

enumerator kVBAT_BandgapRefresh7P8125ms
Bandgap refresh every 7.8125ms.

enumerator kVBAT_BandgapRefresh15P625ms
Bandgap refresh every 15.625ms.

enumerator kVBAT_BandgapRefresh31P25ms
Bandgap refresh every 31.25ms.

enumerator kVBAT_BandgapRefresh62P5ms
Bandgap refresh every 62.5ms.

enum _vbat_bandgap_timer_timeout_period

The enumeration of bandgap timer timeout period.

Values:

enumerator kVBAT_BangapTimerTimeout1s

Bandgap timer timerout every 1s.

enumerator kVBAT_BangapTimerTimeout500ms

Bandgap timer timerout every 500ms.

enumerator kVBAT_BangapTimerTimeout250ms

Bandgap timer timerout every 250ms.

enumerator kVBAT_BangapTimerTimeout125ms

Bandgap timer timerout every 125ms.

enumerator kVBAT_BangapTimerTimeout62P5ms

Bandgap timer timerout every 62.5ms.

enumerator kVBAT_BangapTimerTimeout31P25ms

Bandgap timer timerout every 31.25ms.

typedef enum *_vbat_bandgap_refresh_period* vbat_bandgap_refresh_period_t

The enumeration of bandgap refresh period.

typedef enum *_vbat_bandgap_timer_timeout_period* vbat_bandgap_timer_timeout_period_t

The enumeration of bandgap timer timeout period.

typedef struct *_vbat_fro16k_config* vbat_fro16k_config_t

The structure of internal 16kHz free running oscillator attributes.

VBAT_LDO_READY_TIMEOUT

Max loops to wait for LDO ready.

When configuring the LDO, driver will wait for LDO ready. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

void VBAT_ConfigFRO16k(VBAT_Type *base, const *vbat_fro16k_config_t* *config)

Configure internal 16kHz free running oscillator, including enabel FRO16k, gate FRO16k output.

Parameters

- base – VBAT peripheral base address.
- config – Pointer to vbat_fro16k_config_t structure.

static inline void VBAT_EnableFRO16k(VBAT_Type *base, bool enable)

Enable/disable internal 16kHz free running oscillator.

Parameters

- base – VBAT peripheral base address.
- enable – Used to enable/disable 16kHz FRO.
  - **true** Enable internal 16kHz free running oscillator.
  - **false** Disable internal 16kHz free running oscillator.

static inline bool VBAT_CheckFRO16kEnabled(VBAT_Type *base)

Check if internal 16kHz free running oscillator is enabled.

Parameters

- base – VBAT peripheral base address.

Return values

- true – The internal 16kHz Free running oscillator is enabled.
- false – The internal 16kHz Free running oscillator is enabled.

static inline void VBAT_UngateFRO16k(VBAT_Type *base, bool unGateFRO16k)

Ungate/gate FRO 16kHz output clock to other modules.

Parameters

- base – VBAT peripheral base address.
- unGateFRO16k – Used to gate/ungate FRO 16kHz output.
  - **true** FRO 16kHz output clock to other modules is enabled.
  - **false** FRO 16kHz output clock to other modules is disabled.

static inline void VBAT_LockFRO16kSettings(VBAT_Type *base)

Lock settings of internal 16kHz free running oscillator, please note that if locked 16kHz FRO's settings can not be updated until the next POR.

Note: Please note that the operation to ungate/gate FRO 16kHz output clock can not be locked by this function.

Parameters

- base – VBAT peripheral base address.

status_t VBAT_EnableBandgap(VBAT_Type *base, bool enable)

Enable/disable Bandgap.

Note: The FRO16K must be enabled before enabling the bandgap.

Note: This setting can be locked by VBAT_LockLdoRamSettings() function.

Parameters

- base – VBAT peripheral base address.
- enable – Used to enable/disable bandgap.
  - **true** Enable the bandgap.
  - **false** Disable the bandgap.

Return values

- kStatus_Success – Success to enable/disable the bandgap.
- kStatus_VBAT_Fro16kNotEnabled – Fail to enable the bandgap due to FRO16k is not enabled previously.

static inline bool VBAT_CheckBandgapEnabled(VBAT_Type *base)

Check if bandgap is enabled.

Parameters

- base – VBAT peripheral base address.

Return values

- true – The bandgap is enabled.
- false – The bandgap is disabled.

static inline void VBAT_EnableBandgapRefreshMode(VBAT_Type *base, bool enableRefreshMode)

Enable/disable bandgap low power refresh mode.

Note: This setting can be locked by VBAT_LockLdoRamSettings() function.

Parameters

- base – VBAT peripheral base address.
- enableRefreshMode – Used to enable/disable bandgap low power refresh mode.
  - **true** Enable bandgap low power refresh mode.
  - **false** Disable bandgap low power refresh mode.

status_t VBAT_EnableBackupSRAMRegulator(VBAT_Type *base, bool enable)

Enable/disable Backup RAM Regulator(RAM_LDO).

Note: This setting can be locked by VBAT_LockLdoRamSettings() function.

Parameters

- base – VBAT peripheral base address.
- enable – Used to enable/disable RAM_LDO.
  - **true** Enable backup SRAM regulator.
  - **false** Disable backup SRAM regulator.

Return values

- kStatusSuccess – Success to enable/disable backup SRAM regulator.
- kStatus_VBAT_Fro16kNotEnabled – Fail to enable backup SRAM regulator due to FRO16k is not enabled previously.
- kStatus_VBAT_BandgapNotEnabled – Fail to enable backup SRAM regulator due to the bandgap is not enabled previously.
- kStatus_Timeout – Timeout occurs while waiting completion.

static inline void VBAT_LockLdoRamSettings(VBAT_Type *base)

Lock settings of RAM_LDO, please note that if locked then RAM_LDO's settings can not be updated until the next POR.

Parameters

- base – VBAT peripheral base address.

status_t VBAT_SwitchSRAMPowerByVBAT(VBAT_Type *base)

Switch the SRAM to be powered by VBAT in software mode.

Note: This function can be used to switch the SRAM to the VBAT retention supply at any time, but please note that the SRAM must not be accessed during this time and software must manually invoke VBAT_SwitchSRAMPowerBySocSupply() before accessing the SRAM again.

Parameters

- base – VBAT peripheral base address.

Return values

- kStatusSuccess – Success to Switch SRAM powered by VBAT.
- kStatus_VBAT_Fro16kNotEnabled – Fail to switch SRAM powered by VBAT due to FRO16K not enabled previously.

static inline void VBAT_SwitchSRAMPowerBySocSupply(VBAT_Type *base)

Switch the RAM to be powered by Soc Supply in software mode.

Parameters

- base – VBAT peripheral base address.

static inline void VBAT_EnableSRAMArrayRetained(VBAT_Type *base, bool enable)

Enable/disable SRAM array remains powered from Soc power, when LDO_RAM is disabled.

Parameters

- base – VBAT peripheral base address.
- enable – Used to enable/disable SRAM array power retained.
  - **true** SRAM array is retained when powered from VDD_CORE.
  - **false** SRAM array is not retained when powered from VDD_CORE.

static inline void VBAT_EnableSRAMIsolation(VBAT_Type *base, bool enable)

Enable/disable SRAM isolation.

Parameters

- base – VBAT peripheral base address.
- enable – Used to enable/disable SRAM violation.
  - **true** SRAM will be isolated.
  - **false** SRAM state follows the SoC power modes.

status_t VBAT_EnableBandgapTimer(VBAT_Type *base, bool enable, uint8_t timerIdMask)

Enable/disable Bandgap timer.

Note: The bandgap timer is available when the bandgap is enabled and are clocked by the FRO16k.

Parameters

- base – VBAT peripheral base address.
- enable – Used to enable/disable bandgap timer.
- timerIdMask – The mask of bandgap timer Id, should be the OR'ed value of vbat_bandgap_timer_id_t.

Return values

- kStatus_Success – Success to enable/disable selected bandgap timer.
- kStatus_VBAT_Fro16kNotEnabled – Fail to enable/disable selected bandgap timer due to FRO16k not enabled previously.
- kStatus_VBAT_BandgapNotEnabled – Fail to enable/disable selected bandgap timer due to bandgap not enabled previously.

void VBAT_SetBandgapTimerTimeoutValue(VBAT_Type *base, *vbat_bandgap_timer_timeout_period_t* timeoutPeriod, uint8_t timerIdMask)

Set bandgap timer timeout value.

Parameters

- base – VBAT peripheral base address.
- timeoutPeriod – Bandgap timer timeout value, please refer to vbat_bandgap_timer_timeout_period_t.
- timerIdMask – The mask of bandgap timer Id, should be the OR'ed value of vbat_bandgap_timer_id_t.

static inline uint32_t VBAT_GetStatusFlags(VBAT_Type *base)

Get VBAT status flags.

Parameters

- base – VBAT peripheral base address.

Returns

The asserted status flags, should be the OR'ed value of vbat_status_flag_t.

static inline void VBAT_ClearStatusFlags(VBAT_Type *base, uint32_t mask)

Clear VBAT status flags.

Parameters

- base – VBAT peripheral base address.
- mask – The mask of status flags to be cleared, should be the OR'ed value of vbat_status_flag_t except kVBAT_StatusFlagLdoReady.

static inline void VBAT_EnableInterrupts(VBAT_Type *base, uint32_t mask)

Enable interrupts for the VBAT module, such as POR detect interrupt, Wakeup Pin interrupt and so on.

Parameters

- base – VBAT peripheral base address.
- mask – The mask of interrupts to be enabled, should be the OR'ed value of vbat_interrupt_enable_t.

static inline void VBAT_DisableInterrupts(VBAT_Type *base, uint32_t mask)

Disable interrupts for the VBAT module, such as POR detect interrupt, wakeup pin interrupt and so on.

Parameters

- base – VBAT peripheral base address.
- mask – The mask of interrupts to be disabled, should be the OR'ed value of vbat_interrupt_enable_t.

static inline void VBAT_EnableWakeup(VBAT_Type *base, uint32_t mask)

Enable wakeup for the VBAT module, such as POR detect wakeup, wakeup pin wakeup and so on.

Parameters

- base – VBAT peripheral base address.
- mask – The mask of enumerators in vbat_wakeup_enable_t.

static inline void VBAT_DisableWakeup(VBAT_Type *base, uint32_t mask)

Disable wakeup for VBAT module, such as POR detect wakeup, wakeup pin wakeup and so on.

Parameters

- base – VBAT peripheral base address.
- mask – The mask of enumerators in vbat_wakeup_enable_t.

static inline void VBAT_LockInterruptWakeupSettings(VBAT_Type *base)

Lock VBAT interrupt and wakeup settings, please note that if locked the interrupt and wakeup settings can not be updated until the next POR.

Parameters

- base – VBAT peripheral base address.

FSL_VBAT_DRIVER_VERSION

VBAT driver version 2.1.1.

struct _vbat_fro16k_config

#include <fsl_vbat.h> The structure of internal 16kHz free running oscillator attributes.

Public Members

bool enableFRO16k

Enable/disable internal 16kHz free running oscillator.

bool enableFRO16kOutput

Enable/disable FRO 16k output clock to other modules.

2.67 VREF: Voltage Reference Driver

void VREF_Init(VREF_Type *base, const *vref_config_t* *config)

Enables the clock gate and configures the VREF module according to the configuration structure.

This function must be called before calling all other VREF driver functions, read/write registers, and configurations with user-defined settings. The example below shows how to set up vref_config_t parameters and how to call the VREF_Init function by passing in these parameters.

```
vref_config_t vrefConfig;
VREF_GetDefaultConfig(&vrefConfig);
vrefConfig.bufferMode = kVREF_ModeHighPowerBuffer;
VREF_Init(VREF, &vrefConfig);
```

Parameters

- base – VREF peripheral address.
- config – Pointer to the configuration structure.

void VREF_Deinit(VREF_Type *base)

Stops and disables the clock for the VREF module.

This function should be called to shut down the module. This is an example.

```
vref_config_t vrefUserConfig;
VREF_GetDefaultConfig(&vrefUserConfig);
VREF_Init(VREF, &vrefUserConfig);
...
VREF_Deinit(VREF);
```

Parameters

- base – VREF peripheral address.

void VREF_GetDefaultConfig(*vref_config_t* *config)

Initializes the VREF configuration structure.

This function initializes the VREF configuration structure to default values. This is an example.

```
config->bufferMode = kVREF_ModeHighPowerBuffer;
config->enableInternalVoltageRegulator = true;
config->enableChopOscillator           = true;
config->enableHCBandgap                = true;
config->enableCurvatureCompensation    = true;
config->enableLowPowerBuff             = true;
```

Parameters

- config – Pointer to the initialization structure.

void VREF_SetVrefTrimVal(VREF_Type *base, uint8_t trimValue)

Sets a TRIM value for the accurate 1.0V bandgap output.

This function sets a TRIM value for the reference voltage. It will trim the accurate 1.0V bandgap by 0.5mV each step.

Parameters

- base – VREF peripheral address.
- trimValue – Value of the trim register to set the output reference voltage (maximum 0x3F (6-bit)).

void VREF_SetTrim21Val(VREF_Type *base, uint8_t trim21Value)

Sets a TRIM value for the accurate buffered VREF output.

This function sets a TRIM value for the reference voltage. If buffer mode be set to other values (Buf21 enabled), it will trim the VREF_OUT by 0.1V each step from 1.0V to 2.1V.

Note: When Buf21 is enabled, the value of UTRIM[TRIM2V1] should be ranged from 0b0000 to 0b1011 in order to trim the output voltage from 1.0V to 2.1V, other values will make the VREF_OUT to default value, 1.0V.

Parameters

- base – VREF peripheral address.
- trim21Value – Value of the trim register to set the output reference voltage (maximum 0xF (4-bit)).

uint8_t VREF_GetVrefTrimVal(VREF_Type *base)

Reads the trim value.

This function gets the TRIM value from the UTRIM register. It reads UTRIM[VREFTRIM] (13:8)

Parameters

- base – VREF peripheral address.

Returns
6-bit value of trim setting.

uint8_t VREF_GetTrim21Val(VREF_Type *base)

Reads the VREF 2.1V trim value.

This function gets the TRIM value from the UTRIM register. It reads UTRIM[TRIM2V1] (3:0),

Parameters

- base – VREF peripheral address.

Returns
4-bit value of trim setting.

FSL_VREF_DRIVER_VERSION

Version 2.4.0.

enum _vref_buffer_mode

VREF buffer modes.

Values:

enumerator kVREF_ModeBandgapOnly
Bandgap enabled/standby.

enumerator kVREF_ModeLowPowerBuffer
Low-power buffer mode enabled

enumerator kVREF_ModeHighPowerBuffer
High-power buffer mode enabled

typedef enum *_vref_buffer_mode* vref_buffer_mode_t

VREF buffer modes.

typedef struct *_vref_config* vref_config_t

The description structure for the VREF module.

struct _vref_config

#include <fsl_vref.h> The description structure for the VREF module.

Public Members

vref_buffer_mode_t bufferMode
Buffer mode selection

bool enableInternalVoltageRegulator
Provide additional supply noise rejection.

bool enableChopOscillator
Enable Chop oscillator.

bool enableHCBandgap
Enable High-Accurate bandgap.

bool enableCurvatureCompensation
Enable second order curvature compensation.

bool enableLowPowerBuff
Provides bias current for other peripherals.

2.68 WDOG32: 32-bit Watchdog Timer

void WDOG32_GetDefaultConfig(*wdog32_config_t* *config)

Initializes the WDOG32 configuration structure.

This function initializes the WDOG32 configuration structure to default values. The default values are:

```
wdog32Config->enableWdog32 = true;
wdog32Config->clockSource = kWDOG32_ClockSource1;
wdog32Config->prescaler = kWDOG32_ClockPrescalerDivide1;
wdog32Config->workMode.enableWait = true;
wdog32Config->workMode.enableStop = false;
wdog32Config->workMode.enableDebug = false;
wdog32Config->testMode = kWDOG32_TestModeDisabled;
wdog32Config->enableUpdate = true;
wdog32Config->enableInterrupt = false;
wdog32Config->enableWindowMode = false;
wdog32Config->windowValue = 0U;
wdog32Config->timeoutValue = 0xFFFFU;
```

See also:

wdog32_config_t

Parameters

- config – Pointer to the WDOG32 configuration structure.

status_t WDOG32_Init(WDOG_Type *base, const *wdog32_config_t* *config)

Initializes the WDOG32 module.

This function initializes the WDOG32. To reconfigure the WDOG32 without forcing a reset first, enableUpdate must be set to true in the configuration.

Example:

```
wdog32_config_t config;
WDOG32_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
config.enableUpdate = true;
WDOG32_Init(wdog_base,&config);
```

Note: If there is errata ERR010536 (FSL_FEATURE_WDOG_HAS_ERRATA_010536 defined as 1), then after calling this function, user need delay at least 4 LPO clock cycles before accessing other WDOG32 registers.

Parameters

- base – WDOG32 peripheral base address.
- config – The configuration of the WDOG32.

Return values

- kStatus_Success – The initialization was successful
- kStatus_Timeout – The initialization timed out

status_t WDOG32_Deinit(WDOG_Type *base)

De-initializes the WDOG32 module.

This function shuts down the WDOG32. Ensure that the WDOG_CS.UPDATE is 1, which means that the register update is enabled.

Parameters

- base – WDOG32 peripheral base address.

Return values

- kStatus_Success – The de-initialization was successful
- kStatus_Timeout – The de-initialization timed out

status_t WDOG32_Unlock(WDOG_Type *base)

Unlocks the WDOG32 register written.

This function unlocks the WDOG32 register written.

Before starting the unlock sequence and following the configuration, disable the global interrupts. Otherwise, an interrupt could effectively invalidate the unlock sequence and the WCT may expire. After the configuration finishes, re-enable the global interrupts.

Parameters

- base – WDOG32 peripheral base address

Return values

- kStatus_Success – The unlock sequence was successful
- kStatus_Timeout – The unlock sequence timed out

void WDOG32_Enable(WDOG_Type *base)

Enables the WDOG32 module.

This function writes a value into the WDOG_CS register to enable the WDOG32. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG32 peripheral base address.

void WDOG32_Disable(WDOG_Type *base)

Disables the WDOG32 module.

This function writes a value into the WDOG_CS register to disable the WDOG32. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG32 peripheral base address

void WDOG32_EnableInterrupts(WDOG_Type *base, uint32_t mask)

Enables the WDOG32 interrupt.

This function writes a value into the WDOG_CS register to enable the WDOG32 interrupt. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG32 peripheral base address.
- mask – The interrupts to enable. The parameter can be a combination of the following source if defined:
  - kWDOG32_InterruptEnable

void WDOG32_DisableInterrupts(WDOG_Type *base, uint32_t mask)

Disables the WDOG32 interrupt.

This function writes a value into the WDOG_CS register to disable the WDOG32 interrupt. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG32 peripheral base address.
- mask – The interrupts to disabled. The parameter can be a combination of the following source if defined:
  - kWDOG32_InterruptEnable

static inline uint32_t WDOG32_GetStatusFlags(WDOG_Type *base)

Gets the WDOG32 all status flags.

This function gets all status flags.

Example to get the running flag:

```
uint32_t status;
status = WDOG32_GetStatusFlags(wdog_base) & kWDOG32_RunningFlag;
```

See also:

_wdog32_status_flags_t

- true: related status flag has been set.
- false: related status flag is not set.

Parameters

- base – WDOG32 peripheral base address

Returns

State of the status flag: asserted (true) or not-asserted (false).

void WDOG32_ClearStatusFlags(WDOG_Type *base, uint32_t mask)

Clears the WDOG32 flag.

This function clears the WDOG32 status flag.

Example to clear an interrupt flag:

```
WDOG32_ClearStatusFlags(wdog_base,kWDOG32_InterruptFlag);
```

Parameters

- base – WDOG32 peripheral base address.
- mask – The status flags to clear. The parameter can be any combination of the following values:
  - kWDOG32_InterruptFlag

void WDOG32_SetTimeoutValue(WDOG_Type *base, uint16_t timeoutCount)

Sets the WDOG32 timeout value.

This function writes a timeout value into the WDOG_TOVAL register. The WDOG_TOVAL register is a write-once register. To ensure the reconfiguration fits the timing of WCT, unlock function will be called inline.

Parameters

- base – WDOG32 peripheral base address
- timeoutCount – WDOG32 timeout value, count of WDOG32 clock ticks.

void WDOG32_SetWindowValue(WDOG_Type *base, uint16_t windowValue)

Sets the WDOG32 window value.

This function writes a window value into the WDOG_WIN register. The WDOG_WIN register is a write-once register. Please check the enableUpdate is set to true for calling WDOG32_Init to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG32 peripheral base address.
- windowValue – WDOG32 window value.

static inline void WDOG32_Refresh(WDOG_Type *base)

Refreshes the WDOG32 timer.

This function feeds the WDOG32. This function should be called before the Watchdog timer is in timeout. Otherwise, a reset is asserted.

Parameters

- base – WDOG32 peripheral base address

static inline uint16_t WDOG32_GetCounterValue(WDOG_Type *base)

Gets the WDOG32 counter value.

This function gets the WDOG32 counter value.

Parameters

- base – WDOG32 peripheral base address.

Returns

Current WDOG32 counter value.

WDOG_FIRST_WORD_OF_UNLOCK

First word of unlock sequence

WDOG_SECOND_WORD_OF_UNLOCK

Second word of unlock sequence

WDOG_FIRST_WORD_OF_REFRESH

First word of refresh sequence

WDOG_SECOND_WORD_OF_REFRESH

Second word of refresh sequence

FSL_WDOG32_DRIVER_VERSION

WDOG32 driver version.

enum _wdog32_clock_source

Max loops to wait for WDOG32 unlock sequence complete.

This is the maximum number of loops to wait for the wdog32 unlock sequence to complete. If set to 0, it will wait indefinitely until the unlock sequence is complete.

Max loops to wait for WDOG32 reconfiguration complete.

This is the maximum number of loops to wait for the wdog32 reconfiguration to complete. If set to 0, it will wait indefinitely until the reconfiguration is complete.

Describes WDOG32 clock source.

Values:

enumerator kWDOG32_ClockSource0

Clock source 0

enumerator kWDOG32_ClockSource1

Clock source 1

enumerator kWDOG32_ClockSource2

Clock source 2

enumerator kWDOG32_ClockSource3

Clock source 3

enum _wdog32_clock_prescaler

Describes the selection of the clock prescaler.

Values:

enumerator kWDOG32_ClockPrescalerDivide1

Divided by 1

enumerator kWDOG32_ClockPrescalerDivide256

Divided by 256

enum _wdog32_test_mode

Describes WDOG32 test mode.

Values:

enumerator kWDOG32_TestModeDisabled

Test Mode disabled

enumerator kWDOG32_UserModeEnabled

User Mode enabled

enumerator kWDOG32_LowByteTest

Test Mode enabled, only low byte is used

enumerator kWDOG32_HighByteTest

Test Mode enabled, only high byte is used

enum _wdog32_interrupt_enable_t

WDOG32 interrupt configuration structure.

This structure contains the settings for all of the WDOG32 interrupt configurations.

Values:

enumerator kWDOG32_InterruptEnable

Interrupt is generated before forcing a reset

enum __wdog32_status_flags_t

WDOG32 status flags.

This structure contains the WDOG32 status flags for use in the WDOG32 functions.

Values:

enumerator kWDOG32_RunningFlag

Running flag, set when WDOG32 is enabled

enumerator kWDOG32_InterruptFlag

Interrupt flag, set when interrupt occurs

typedef enum *_wdog32_clock_source* wdog32_clock_source_t

Max loops to wait for WDOG32 unlock sequence complete.

This is the maximum number of loops to wait for the wdog32 unlock sequence to complete. If set to 0, it will wait indefinitely until the unlock sequence is complete.

Max loops to wait for WDOG32 reconfiguration complete.

This is the maximum number of loops to wait for the wdog32 reconfiguration to complete. If set to 0, it will wait indefinitely until the reconfiguration is complete.

Describes WDOG32 clock source.

typedef enum *_wdog32_clock_prescaler* wdog32_clock_prescaler_t

Describes the selection of the clock prescaler.

typedef struct *_wdog32_work_mode* wdog32_work_mode_t

Defines WDOG32 work mode.

typedef enum *_wdog32_test_mode* wdog32_test_mode_t

Describes WDOG32 test mode.

typedef struct *_wdog32_config* wdog32_config_t

Describes WDOG32 configuration structure.

struct __wdog32_work_mode

#include <fsl_wdog32.h> Defines WDOG32 work mode.

Public Members

bool enableWait

Enables or disables WDOG32 in wait mode

bool enableStop

Enables or disables WDOG32 in stop mode

bool enableDebug

Enables or disables WDOG32 in debug mode

struct __wdog32_config

#include <fsl_wdog32.h> Describes WDOG32 configuration structure.

Public Members

bool enableWdog32

Enables or disables WDOG32

wdog32_clock_source_t clockSource

Clock source select

wdog32_clock_prescaler_t prescaler
Clock prescaler value

wdog32_work_mode_t workMode
Configures WDOG32 work mode in debug stop and wait mode

wdog32_test_mode_t testMode
Configures WDOG32 test mode

bool enableUpdate
Update write-once register enable

bool enableInterrupt
Enables or disables WDOG32 interrupt

bool enableWindowMode
Enables or disables WDOG32 window mode

uint16_t windowValue
Window value

uint16_t timeoutValue
Timeout value

2.69 WUU: Wakeup Unit driver

void WUU_SetExternalWakeUpPinsConfig(WUU_Type *base, uint8_t pinIndex, const *wuu_external_wakeup_pin_config_t* *config)

Enables and Configs External WakeUp Pins.

This function enables/disables the external pin as wakeup input. What's more this function configs pins options, including edge detection wakeup event and operate mode.

Parameters

- base – MUU peripheral base address.
- pinIndex – The index of the external input pin. See Reference Manual for the details.
- config – Pointer to wuu_external_wakeup_pin_config_t structure.

void WUU_ClearExternalWakeupPinsConfig(WUU_Type *base, uint8_t pinIndex)

Disable and clear external wakeup pin settings.

Parameters

- base – MUU peripheral base address.
- pinIndex – The index of the external input pin.

static inline uint32_t WUU_GetExternalWakeUpPinsFlag(WUU_Type *base)

Gets External Wakeup pin flags.

This function return the external wakeup pin flags.

Parameters

- base – WUU peripheral base address.

Returns
Wakeup flags for all external wakeup pins.

static inline void WUU_ClearExternalWakeUpPinsFlag(WUU_Type *base, uint32_t mask)

Clears External WakeUp Pin flags.

This function clears external wakeup pins flags based on the mask.

Parameters

- base – WUU peripheral base address.
- mask – The mask of Wakeup pin index to be cleared.

void WUU_SetInternalWakeUpModulesConfig(WUU_Type *base, uint8_t moduleIndex, *wuu_internal_wakeup_module_event_t* event)

Config Internal modules' event as the wake up soures.

This function configs the internal modules event as the wake up sources.

Parameters

- base – WUU peripheral base address.
- moduleIndex – The selected internal module. See the Reference Manual for the details.
- event – Select interrupt or DMA/Trigger of the internal module as the wake up source.

void WUU_ClearInternalWakeUpModulesConfig(WUU_Type *base, uint8_t moduleIndex, *wuu_internal_wakeup_module_event_t* event)

Disable an on-chip internal modules' event as the wakeup sources.

Parameters

- base – WUU peripheral base address.
- moduleIndex – The selected internal module. See the Reference Manual for the details.
- event – The event(interrupt or DMA/trigger) of the internal module to disable.

void WUU_SetPinFilterConfig(WUU_Type *base, uint8_t filterIndex, const *wuu_pin_filter_config_t* *config)

Configs and Enables Pin filters.

This function configs Pin filter, including pin select, filer operate mode filer wakeup event and filter edge detection.

Parameters

- base – WUU peripheral base address.
- filterIndex – The index of the pin filer.
- config – Pointer to wuu_pin_filter_config_t structure.

bool WUU_GetPinFilterFlag(WUU_Type *base, uint8_t filterIndex)

Gets the pin filter configuration.

This function gets the pin filter flag.

Parameters

- base – WUU peripheral base address.
- filterIndex – A pin filter index, which starts from 1.

Returns

True if the flag is a source of the existing low-leakage power mode.

void WUU_ClearPinFilterFlag(WUU_Type *base, uint8_t filterIndex)

Clears the pin filter configuration.

This function clears the pin filter flag.

Parameters

- base – WUU peripheral base address.
- filterIndex – A pin filter index to clear the flag, starting from 1.

bool WUU_GetExternalWakeupPinFlag(WUU_Type *base, uint32_t pinIndex)

brief Gets the external wakeup source flag.

This function checks the external pin flag to detect whether the MCU is woken up by the specific pin.

param base WUU peripheral base address. param pinIndex A pin index, which starts from 0. return True if the specific pin is a wakeup source.

void WUU_ClearExternalWakeupPinFlag(WUU_Type *base, uint32_t pinIndex)

brief Clears the external wakeup source flag.

This function clears the external wakeup source flag for a specific pin.

param base WUU peripheral base address. param pinIndex A pin index, which starts from 0.

FSL_WUU_DRIVER_VERSION

Defines WUU driver version 2.4.1.

enum _wuu_external_pin_edge_detection

External WakeUp pin edge detection enumeration.

Values:

enumerator kWUU_ExternalPinDisable

External input Pin disabled as wake up input.

enumerator kWUU_ExternalPinRisingEdge

External input Pin enabled with the rising edge detection.

enumerator kWUU_ExternalPinFallingEdge

External input Pin enabled with the falling edge detection.

enumerator kWUU_ExternalPinAnyEdge

External input Pin enabled with any change detection.

enum _wuu_external_wakeup_pin_event

External input wake up pin event enumeration.

Values:

enumerator kWUU_ExternalPinInterrupt

External input Pin configured as interrupt.

enumerator kWUU_ExternalPinDMARequest

External input Pin configured as DMA request.

enumerator kWUU_ExternalPinTriggerEvent

External input Pin configured as Trigger event.

enum _wuu_external_wakeup_pin_mode

External input wake up pin mode enumeration.

Values:

enumerator kWUU_ExternalPinActiveDSPD

External input Pin is active only during Deep Sleep/Power Down Mode. NOTE: This enumerations has been deprecated, please switch to kWUU_ExternalPinActiveLowLeakage.

enumerator kWUU_ExternalPinActiveLowLeakageMode

External input Pin is active only during low-leakage power modes.

enumerator kWUU_ExternalPinActiveAlways

External input Pin is active during all power modes.

enum _wuu_internal_wakeup_module_event

Internal module wake up event enumeration.

Values:

enumerator kWUU_InternalModuleInterrupt

Internal modules' interrupt as a wakeup source.

enumerator kWUU_InternalModuleDMATrigger

Internal modules' DMA/Trigger as a wakeup source.

enum _wuu_filter_edge

Pin filter edge enumeration.

Values:

enumerator kWUU_FilterDisabled

Filter disabled.

enumerator kWUU_FilterPosedgeEnable

Filter posedge detect enabled.

enumerator kWUU_FilterNegedgeEnable

Filter negedge detect enabled.

enumerator kWUU_FilterAnyEdge

Filter any edge detect enabled.

enum _wuu_filter_event

Pin Filter event enumeration.

Values:

enumerator kWUU_FilterInterrupt

Filter output configured as interrupt.

enumerator kWUU_FilterDMARequest

Filter output configured as DMA request.

enumerator kWUU_FilterTriggerEvent

Filter output configured as Trigger event.

enum _wuu_filter_mode

Pin filter mode enumeration.

Values:

enumerator kWUU_FilterActiveDSPD

External input pin filter is active only during Deep Sleep/Power Down Mode. NOTE: This enumerations has been deprecated, please switch to kWUU_FilterActiveLowLeakage.

enumerator kWUU_FilterActiveLowLeakageMode
External input pin filter is active only during low-leakage power modes.

enumerator kWUU_FilterActiveAlways
External input Pin filter is active during all power modes.

typedef enum *_wuu_external_pin_edge_detection* wuu_external_pin_edge_detection_t
External WakeUp pin edge detection enumeration.

typedef enum *_wuu_external_wakeup_pin_event* wuu_external_wakeup_pin_event_t
External input wake up pin event enumeration.

typedef enum *_wuu_external_wakeup_pin_mode* wuu_external_wakeup_pin_mode_t
External input wake up pin mode enumeration.

typedef enum *_wuu_internal_wakeup_module_event* wuu_internal_wakeup_module_event_t
Internal module wake up event enumeration.

typedef enum *_wuu_filter_edge* wuu_filter_edge_t
Pin filter edge enumeration.

typedef enum *_wuu_filter_event* wuu_filter_event_t
Pin Filter event enumeration.

typedef enum *_wuu_filter_mode* wuu_filter_mode_t
Pin filter mode enumeration.

typedef struct *_wuu_external_wakeup_pin_config* wuu_external_wakeup_pin_config_t
External WakeUp pin configuration.

typedef struct *_wuu_pin_filter_config* wuu_pin_filter_config_t
Pin Filter configuration.

struct __wuu_external_wakeup_pin_config
#include <fsl_wuu.h> External WakeUp pin configuration.

Public Members

wuu_external_pin_edge_detection_t edge
External Input pin edge detection.

wuu_external_wakeup_pin_event_t event
External Input wakeup Pin event

wuu_external_wakeup_pin_mode_t mode
External Input wakeup Pin operate mode.

struct __wuu_pin_filter_config
#include <fsl_wuu.h> Pin Filter configuration.

Public Members

uint32_t pinIndex
The index of wakeup pin to be muxxed into filter.

wuu_filter_edge_t edge
The edge of the pin digital filter.

wuu_filter_event_t event
The event of the filter output.

wuu_filter_mode_t mode
The mode of the filter operate.

Chapter 3

Middleware

3.1 Motor Control

3.1.1 FreeMASTER

Communication Driver User Guide

Introduction

What is FreeMASTER? FreeMASTER is a PC-based application developed by NXP for NXP customers. It is a versatile tool usable as a real-time monitor, visualization tool, and a graphical control panel of embedded applications based on the NXP processing units.

This document describes the embedded-side software driver which implements an interface between the application and the host PC. The interface covers the following communication:

- **Serial** UART communication either over plain RS232 interface or more typically over a USB-to-Serial either external or built in a debugger probe.
- **USB** direct connection to target microcontroller
- **CAN bus**
- **TCP/IP network** wired or WiFi
- **Segger J-Link RTT**
- **JTAG** debug port communication
- ...and all of the above also using a **Zephyr** generic drivers.

The driver also supports so-called "packet-driven BDM" interface which enables a protocol-based communication over a debugging port. The BDM stands for Background Debugging Module and its physical implementation is different on each platform. Some platforms leverage a semi-standard JTAG interface, other platforms provide a custom implementation called BDM. Regardless of the name, this debugging interface enables non-intrusive access to the memory space while the target CPU is running. For basic memory read and write operations, there is no communication driver required on the target when communicating with the host PC. Use this driver to get more advanced FreeMASTER protocol features over the BDM interface. The driver must be configured for the packet-driven BDM mode, in which the host PC uses the debugging interface to write serial command frames directly to the target memory buffer. The same method is then used to read response frames from that memory buffer.

Similar to "packet-driven BDM", the FreeMASTER also supports a communication over [J-Link RTT]((https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/) interface defined by SEGGER Microcontroller GmbH for ARM CortexM-based microcontrollers. This method also uses JTAG physical interface and enables high-speed real time communication to run over the same channel as used for application debugging.

Driver version 3 This document describes version 3 of the FreeMASTER Communication Driver. This version features the implementation of the new Serial Protocol, which significantly extends the features and security of its predecessor. The new protocol internal number is v4 and its specification is available in the documentation accompanying the driver code.

Driver V3 is deployed to modern 32-bit MCU platforms first, so the portfolio of supported platforms is smaller than for the previous V2 versions. It is recommended to keep using the V2 driver for legacy platforms, such as S08, S12, ColdFire, or Power Architecture. Reach out to FreeMASTER community or to the local NXP representative with requests for more information or to port the V3 driver to legacy MCU devices.

Thanks to a layered approach, the new driver simplifies the porting of the driver to new UART, CAN or networking communication interfaces significantly. Users are encouraged to port the driver to more NXP MCU platforms and contribute the code back to NXP for integration into future releases. Existing code and low-level driver layers may be used as an example when porting to new targets.

Note: Using the FreeMASTER tool and FreeMASTER Communication Driver is only allowed in systems based on NXP microcontroller or microprocessor unit. Use with non-NXP MCU platforms is **not permitted** by the license terms.

Target platforms The driver implementation uses the following abstraction mechanisms which simplify driver porting and supporting new communication modules:

- **General CPU Platform** (see source code in the `src/platforms` directory). The code in this layer is only specific to native data type sizes and CPU architectures (for example; alignment-aware memory copy routines). This driver version brings two generic implementations of 32-bit platforms supporting both little-endian and big-endian architectures. There are also implementations customized for the 56F800E family of digital signal controllers and S12Z MCUs. **Zephyr** is treated as a specific CPU platform as it brings unified user configuration (Kconfig) and generic hardware device drivers. With Zephyr, the transport layer and low-level communication layers described below are configured automatically using Kconfig and Device Tree technologies.
- **Transport Communication Layer** - The Serial, CAN, Networking, PD-BDM, and other methods of transport logic are implemented as a driver layer called FMSTR_TRANSPORT with a uniform API. A support of the Network transport also extends single-client modes of operation which are native for Serial, USB and CAN by a concept of multiple client sessions.
- **Low-level Communication Driver** - Each type of transport further defines a low-level API used to access the physical communication module. For example, the Serial transport defines a character-oriented API implemented by different serial communication modules like UART, LPUART, USART, and also USB-CDC. Similarly, the CAN transport defines a message-oriented API implemented by the FlexCAN or MCAN modules. Moreover, there are multiple different implementations for the same kind of communication peripherals. The difference between the implementation is in the way the low-level hardware registers are accessed. The *mcuxsdk* folder contains implementations which use MCUXpresso SDK drivers. These drivers should be used in applications based on the NXP MCUXpresso SDK. The "ampsdk" drivers target automotive-specific MCUs and their respective SDKs. The "dreg" implementations use a plain C-language access to hardware register addresses which makes it a universal and the most portable solution. In this case, users are encouraged to add more drivers for other communication modules or other respective SDKs and contribute the code back to NXP for integration.

The low-level drivers defined for the Networking transport enable datagram-oriented UDP and stream TCP communication. This implementation is demonstrated using the lwIP software stack but shall be portable to other TCP/IP stacks. It may sound surprisingly, but also the Segger J-Link RTT communication driver is linked to the Networking transport (RTT is stream oriented communication handled similarly to TCP).

Replacing existing drivers For all supported platforms, the driver described in this document replaces the V2 implementation and also older driver implementations that were available separately for individual platforms (PC Master SCI drivers).

Clocks, pins, and peripheral initialization The FreeMASTER communication driver is only responsible for runtime processing of the communication and must be integrated with an user application code to function properly. The user application code is responsible for general initialization of clock sources, pin multiplexers, and peripheral registers related to the communication speed. Such initialization should be done before calling the `FMSTR_Init` function.

It is recommended to develop the user application using one of the Software Development Kits (SDKs) available from third parties or directly from NXP, such as MCUXpresso SDK, MCUXpresso IDE, and related tools. This approach simplifies the general configuration process significantly.

MCUXpresso SDK The MCUXpresso SDK is a software package provided by NXP which contains the device initialization code, linker files, and software drivers with example applications for the NXP family of MCUs. The MCUXpresso Config Tools may be used to generate the clock-setup and pin-multiplexer setup code suitable for the selected processor.

The MCUXpresso SDK also contains this FreeMASTER communication driver as a "middleware" component which may be downloaded along with the example applications from https://mcuxpresso.nxp.com/en/welcome.

MCUXpresso SDK on GitHub The FreeMASTER communication driver is also released as one of the middleware components of the MCUXpresso SDK on the GitHub. This release enables direct integration of the FreeMASTER source code Git repository into a target applications including Zephyr applications.

Related links:

- The official FreeMASTER middleware repository.
- Online version of this document

FreeMASTER in Zephyr The FreeMASTER middleware repository can be used with MCUXpresso SDK as well as a Zephyr module. Zephyr-specific samples which include examples of Kconfig and Device Tree configurations for Serial, USB and Network communications are available in separate repository. West manifest in this sample repository fetches the full Zephyr package including the FreeMASTER middleware repository used as a Zephyr module.

Example applications

MCUX SDK Example applications There are several example applications available for each supported MCU platform.

- **fmstr_uart** demonstrates a plain serial transmission, typically connecting to a computer's physical or virtual COM port. The typical transmission speed is 115200 bps.

- **fmstr_can** demonstrates CAN bus communication. This requires a suitable CAN interface connected to the computer and interconnected with the target MCU using a properly terminated CAN bus. The typical transmission speed is 500 kbps. A FreeMASTER-over-CAN communication plug-in must be used.
- **fmstr_usb_cdc** uses an on-chip USB controller to implement a CDC communication class. It is connected directly to a computer's USB port and creates a virtual COM port device. The typical transmission speed is above 1 Mbps.
- **fmstr_net** demonstrates the Network communication over UDP or TCP protocol. Existing examples use lwIP stack to implement the communication, but in general, it shall be possible to use any other TCP/IP stack to achieve the same functionality.
- **fmstr_wifi** is the fmstr_net application modified to use a WiFi network interface instead of a wired Ethernet connection.
- **fmstr_rtt** demonstrates the communication over SEGGER J-Link RTT interface. Both fmstr_net and fmstr_rtt examples require the FreeMASTER TCP/UDP communication plug-in to be used on the PC host side.
- **fmstr_eonce** uses the real-time data unit on the JTAG EOnCE module of the 56F800E family to implement pseudo-serial communication over the JTAG port. The typical transmission speed is around 10 kbps. This communication requires FreeMASTER JTAG/EOnCE communication plug-in.
- **fmstr_pdbdm** uses JTAG or BDM debugging interface to access the target RAM directly while the CPU is running. Note that such approach can be used with any MCU application, even without any special driver code. The computer reads from and writes into the RAM directly without CPU intervention. The Packet-Driven BDM (PD-BDM) communication uses the same memory access to exchange command and response frames. With PD-BDM, the FreeMASTER tool is able to go beyond basic memory read/write operations and accesses also advanced features like Recorder, TSA, or Pipes. The typical transmission speed is around 10 kbps. A PD-BDM communication plug-in must be used in FreeMASTER and configured properly for the selected debugging interface. Note that this communication cannot be used while a debugging interface is used by a debugger session.
- **fmstr_any** is a special example application which demonstrates how the NXP MCUXpresso Config Tools can be used to configure pins, clocks, peripherals, interrupts, and even the FreeMASTER "middleware" driver features in a graphical and user friendly way. The user can switch between the Serial, CAN, and other ways of communication and generate the required initialization code automatically.

Zephyr sample spplications Zephyr sample applications demonstrate Kconfig and Device Tree configuration which configure the FreeMASTER middleware module for a selected communication option (Serial, CAN, Network or RTT).

Refer to *readme.md* files in each sample directory for description of configuration options required to implement FreeMASTER connectivity.

Description

This section shows how to add the FreeMASTER Communication Driver into application and how to configure the connection to the FreeMASTER visualization tool.

Features The FreeMASTER driver implements the FreeMASTER protocol V4 and provides the following features which may be accessed using the FreeMASTER visualization tool:

- Read/write access to any memory location on the target.
- Optional password protection of the read, read/write, and read/write/flash access levels.

- Atomic bit manipulation on the target memory (bit-wise write access).
- Optimal size-aligned access to memory which is also suitable to access the peripheral register space.
- Oscilloscope access—real-time access to target variables. The sample rate may be limited by the communication speed.
- Recorder— access to the fast transient recorder running on the board as a part of the FreeMASTER driver. The sample rate is only limited by the MCU CPU speed. The length of the data recorded depends on the amount of available memory.
- Multiple instances of Oscilloscopes and Recorders without the limitation of maximum number of variables.
- Application commands—high-level message delivery from the PC to the application.
- TSA tables—describing the data types, variables, files, or hyperlinks exported by the target application. The TSA newly supports also non-memory mapped resources like external EEPROM or SD Card files.
- Pipes—enabling the buffered stream-oriented data exchange for a general-purpose terminal-like communication, diagnostic data streaming, or other data exchange.

The FreeMASTER driver features:

- Full FreeMASTER protocol V4 implementation with a new V4 style of CRC used.
- Layered approach supporting Serial, CAN, Network, PD-BDM, and other transports.
- Layered low-level Serial transport driver architecture enabling to select UART, LPUART, USART, and other physical implementations of serial interfaces, including USB-CDC.
- Layered low-level CAN transport driver architecture enabling to select FlexCAN, msCAN, MCAN, and other physical implementations of the CAN interface.
- Layered low-level Networking transport enabling to select TCP, UDP or J-Link RTT communication.
- TSA support to write-protect memory regions or individual variables and to deny the access to the unsafe memory.
- The pipe callback handlers are invoked whenever new data is available for reading from the pipe.
- Two Serial Single-Wire modes of operation are enabled. The “external” mode has the RX and TX shorted on-board. The “true” single-wire mode interconnects internally when the MCU or UART modules support it.

The following sections briefly describe all FreeMASTER features implemented by the driver. See the PC-based FreeMASTER User Manual for more details on how to use the features to monitor, tune, or control an embedded application.

Board Detection The FreeMASTER protocol V4 defines the standard set of configuration values which the host PC tool reads to identify the target and to access other target resources properly. The configuration includes the following parameters:

- Version of the driver and the version of the protocol implemented.
- MTU as the Maximum size of the Transmission Unit (for example; communication buffer size).
- Application name, description, and version strings.
- Application build date and time as a string.
- Target processor byte ordering (little/big endian).
- Protection level that requires password authentication.

- Number of the Recorder and Oscilloscope instances.
- RAM Base Address for optimized memory access commands.

Memory Read This basic feature enables the host PC to read any data memory location by specifying the address and size of the required memory area. The device response frame must be shorter than the MTU to fit into the outgoing communication buffer. To read a device memory of any size, the host uses the information retrieved during the Board Detection and splits the large-block request to multiple partial requests.

The driver uses size-aligned operations to read the target memory (for example; uses proper read-word instruction when an address is aligned to 4 bytes).

Memory Write Similarly to the Memory Read operation, the Memory Write feature enables to write to any RAM memory location on the target device. A single write command frame must be shorter than the MTU to fit into the target communication buffer. Larger requests must be split into smaller ones.

The driver uses size-aligned operations to write to the target memory (for example; uses proper write-word instruction when an address is aligned to 4 bytes).

Masked Memory Write To implement the write access to a single bit or a group of bits of target variables, the Masked Memory Write feature is available in the FreeMASTER protocol and it is supported by the driver using the Read-Modify-Write approach.

Be careful when writing to bit fields of volatile variables that are also modified in an application interrupt. The interrupt may be serviced in the middle of a read-modify-write operation and it may cause data corruption.

Oscilloscope The protocol and driver enables any number of variables to be read at once with a single request from the host. This feature is called Oscilloscope and the FreeMASTER tool uses it to display a real-time graph of variable values.

The driver can be configured to support any number of Oscilloscope instances and enable simultaneously running graphs to be displayed on the host computer screen.

Recorder The protocol enables the host to select target variables whose values are then periodically recorded into a dedicated on-board memory buffer. After such data sampling stops (either on a host request or by evaluating a threshold-crossing condition), the data buffer is downloaded to the host and displayed as a graph. The data sampling rate is not limited by the speed of the communication line, so it enables displaying the variable transitions in a very high resolution.

The driver can be configured to support multiple Recorder instances and enable multiple recorder graphs to be displayed on the host screen. Having multiple recorders also enables setting the recording point differently for each instance. For example; one instance may be recording data in a general timer interrupt while another instance may record at a specific control algorithm time in the PWM interrupt.

TSA With the TSA feature, data types and variables can be described directly in the application source code. Such information is later provided to the FreeMASTER tool which may use it instead of reading symbol data from the application ELF executable file.

The information is encoded as so-called TSA tables which become direct part of the application code. The TSA tables contain descriptors of variables that shall be visible to the host tool. The descriptors can describe the memory areas by specifying the address and size of the memory

block or more conveniently using the C variable names directly. Different set of TSA descriptors can be used to encode information about the structure types, unions, enumerations, or arrays.

The driver also supports special types of TSA table entries to describe user resources like external EEPROM and SD Card files, memory-mapped files, virtual directories, web URL hyperlinks, and constant enumerations.

TSA Safety When the TSA is enabled in the application, the TSA Safety can be enabled and validate the memory accesses directly by the embedded-side driver. When the TSA Safety is turned on, any memory request received from the host is validated and accepted only if it belongs to a TSA-described object. The TSA entries can be declared as Read-Write or Read-Only so that the driver can actively deny the write access to the Read-Only objects.

Application commands The Application Commands are high-level messages that can be delivered from the PC Host to the embedded application for further processing. The embedded application can either poll the status, or be called back when a new Application Command arrives to be processed. After the embedded application acknowledges that the command is handled, the host receives the Result Code and reads the other return data from memory. Both the Application Commands and the Result Codes are specific to a given application and it is user's responsibility to define them. The FreeMASTER protocol and the FreeMASTER driver only implement the delivery channel and a set of API calls to enable the Application Command processing in general.

Pipes The Pipes enable buffered and stream-oriented data exchange between the PC Host and the target application. Any pipe can be written to and read from at both ends (either on the PC or the MCU). The data transmission is acknowledged using the special FreeMASTER protocol commands. It is guaranteed that the data bytes are delivered from the writer to the reader in a proper order and without losses.

Serial single-wire operation The MCU Serial Communication Driver natively supports normal dual-wire operation. Because the protocol is half-duplex only, the driver can also operate in two single-wire modes:

- "External" single-wire operation where the Receiver and Transmitter pins are shorted on the board. This mode is supported by default in the MCU driver because the Receiver and Transmitter units are enabled or disabled whenever needed. It is also easy to extend this operation for the RS485 communication.
- "True" single-wire mode which uses only a single pin and the direction switching is made by the UART module. This mode of operation must be enabled by defining the FMSTR_SERIAL_SINGLEWIRE configuration option.

Multi-session support With networking interface it is possible for multiple clients to access the target MCU simultaneously. Reading and writing of target memory is processed atomically so there is no risk of data corruption. The state-full resources such as Recorders or Oscilloscopes are locked to a client session upon first use and access is denied to other clients until lock is released..

Zephyr-specific

Dedicated communication task FreeMASTER communication may run isolated in a dedicated task. The task automates the FMSTR_Init and FMSTR_Poll calls together with periodic activities enabling the FreeMASTER UI to fetch information about tasks and CPU utilization. The task can be started automatically or manually, and it must be assigned a priority to be able to react on interrupts and other communication events. Refer to Zephyr FreeMASTER sample applications which all use this communication task.

Zephyr shell and logging over FreeMASTER pipe FreeMASTER implements a shell backend which may use FreeMASTER pipe as a I/O terminal and logging output. Refer to Zephyr FreeMASTER sample applications which all use this feature.

Automatic TSA tables TSA tables can be declared as "automatic" in Zephyr which make them automatically registered in the table list. This may be very useful when there are many TSA tables or when the tables are defined in different (often unrelated) libraries linked together. In this case user does not need to build a list of all tables manually.

Driver files The driver source files can be found in a top-level src folder, further divided into the sub-folders:

- ***src/platforms*** platform-specific folder—one folder exists for each supported processor platform (for example; 32-bit Little Endian platform). Each such folder contains a platform header file with data types and a code which implements the potentially platform-specific operations, such as aligned memory access.
- ***src/common*** folder—contains the common driver source files shared by the driver for all supported platforms. All the *.c* files must be added to the project, compiled, and linked together with the application.
  - *freemaster.h* - master driver header file, which declares the common data types, macros, and prototypes of the FreeMASTER driver API functions.
  - *freemaster_cfg.h.example* - this file can serve as an example of the FreeMASTER driver configuration file. Save this file into a project source code folder and rename it to *freemaster_cfg.h*. The FreeMASTER driver code includes this file to get the project-specific configuration options and to optimize the compilation of the driver.
  - *freemaster_defcfg.h* - defines the default values for each FreeMASTER configuration option if the option is not set in the *freemaster_cfg.h* file.
  - *freemaster_protocol.h* - defines the FreeMASTER protocol constants used internally by the driver.
  - *freemaster_protocol.c* - implements the FreeMASTER protocol decoder and handles the basic Get Configuration Value, Memory Read, and Memory Write commands.
  - *freemaster_rec.c* - handles the Recorder-specific commands and implements the Recorder sampling and triggering routines. When the Recorder is disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.
  - *freemaster_scope.c* - handles the Oscilloscope-specific commands. If the Oscilloscope is disabled by the FreeMASTER driver configuration file, this file compiles as void.
  - *freemaster_pipes.c* - implements the Pipes functionality when the Pipes feature is enabled.
  - *freemaster_appcmd.c* - handles the communication commands used to deliver and execute the Application Commands within the context of the embedded application. When the Application Commands are disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.

  - *freemaster_tsa.c* - handles the commands specific to the TSA feature. This feature enables the FreeMASTER host tool to obtain the TSA memory descriptors declared in the embedded application. If the TSA is disabled by the FreeMASTER driver configuration file, this file compiles as void.
  - *freemaster_tsa.h* - contains the declaration of the macros used to define the TSA memory descriptors. This file is indirectly included into the user application code (via *freemaster.h*).
  - *freemaster_sha.c* - implements the SHA-1 hash code used in the password authentication algorithm.
  - *freemaster_private.h* - contains the declarations of functions and data types used internally in the driver. It also contains the C pre-processor statements to perform the compile-time verification of the user configuration provided in the *freemaster_cfg.h* file.
  - *freemaster_serial.c* - implements the serial protocol logic including the CRC, FIFO queuing, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a character-oriented API exported by the specific low-level driver.
  - *freemaster_serial.h* - defines the low-level character-oriented Serial API.
  - *freemaster_can.c* - implements the CAN protocol logic including the CAN message preparation, signalling using the first data byte in the CAN frame, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a message-oriented API exported by the specific low-level driver.
  - *freemaster_can.h* - defines the low-level message-oriented CAN API.
  - *freemaster_net.c* - implements the Network protocol transport logic including multiple session management code.
  - *freemaster_net.h* - definitions related to the Network transport.
  - *freemaster_pdbdm.c* - implements the packet-driven BDM communication buffer and other communication-related operations.
  - *freemaster_utils.c* - aligned memory copy routines, circular buffer management and other utility functions
  - *freemaster_utils.h* - definitions related to utility code.
- ***src/drivers/[sdk]/serial*** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
  - *freemaster_serial_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the UART, LPUART, USART, and other kinds of Serial communication modules.
- ***src/drivers/[sdk]/can*** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
  - *freemaster_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the FlexCAN, msCAN, MCAN, and other kinds of CAN communication modules.
- ***src/drivers/[sdk]/network*** - contains low-level code adapting the FreeMASTER Network transport to an underlying TCP/IP or RTT stack.
  - *freemaster_net_lwip_tcp.c* and *_udp.c* - default networking implementation of TCP and UDP transports using lwIP stack.
  - *freemaster_net_segger_rtt.c* - implementation of network transport using Segger J-Link RTT interface

Driver configuration The driver is configured using a single header file (*freemaster_cfg.h*). Create this file and save it together with other project source files before compiling the driver code. All FreeMASTER driver source files include the *freemaster_cfg.h* file and use the macros defined here for the conditional and parameterized compilation. The C compiler must locate the configuration file when compiling the driver files. Typically, it can be achieved by putting this file into a folder where the other project-specific included files are stored.

As a starting point to create the configuration file, get the *freemaster_cfg.h.example* file, rename it to *freemaster_cfg.h*, and save it into the project area.

Note: It is NOT recommended to leave the *freemaster_cfg.h* file in the FreeMASTER driver source code folder. The configuration file must be placed at a project-specific location, so that it does not affect the other applications that use the same driver.

Configurable items This section describes the configuration options which can be defined in *freemaster_cfg.h*.

Interrupt modes

```
#define FMSTR_LONG_INTR   [0|1]
#define FMSTR_SHORT_INTR  [0|1]
#define FMSTR_POLL_DRIVEN [0|1]
```

Value Type boolean (0 or 1)

Description Exactly one of the three macros must be defined to non-zero. The others must be defined to zero or left undefined. The non-zero-defined constant selects the interrupt mode of the driver. See *Driver interrupt modes*.

- FMSTR_LONG_INTR — long interrupt mode
- FMSTR_SHORT_INTR — short interrupt mode
- FMSTR_POLL_DRIVEN — poll-driven mode

Note: Some options may not be supported by all communication interfaces. For example, the FMSTR_SHORT_INTR option is not supported by the USB_CDC interface.

Protocol transport

```
#define FMSTR_TRANSPORT [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER source code. Specify one of existing instances to make use of the protocol transport.

Description Use one of the pre-defined constants, as implemented by the FreeMASTER code. The current driver supports the following transports:

- **FMSTR_SERIAL** - serial communication protocol
- **FMSTR_CAN** - using CAN communication
- **FMSTR_PDBDM** - using packet-driven BDM communication
- **FMSTR_NET** - network communication using TCP or UDP protocol

Serial transport This section describes configuration parameters used when serial transport is used:

```
#define FMSTR_TRANSPORT FMSTR_SERIAL
```

FMSTR_SERIAL_DRV Select what low-level driver interface will be used when implementing the Serial communication.

```
#define FMSTR_SERIAL_DRV [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing serial driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/serial* implementation):

- **FMSTR_SERIAL_MCUX_UART** - UART driver
- **FMSTR_SERIAL_MCUX_LPUART** - LPUART driver
- **FMSTR_SERIAL_MCUX_USART** - USART driver
- **FMSTR_SERIAL_MCUX_MINIUSART** - miniUSART driver
- **FMSTR_SERIAL_MCUX_QSCI** - DSC QSCI driver
- **FMSTR_SERIAL_MCUX_USB** - USB/CDC class driver (also see code in the */support/mcuxsdk_usb* folder)
- **FMSTR_SERIAL_56F800E_EONCE** - DSC JTAG EOnCE driver

Other SDKs or BSPs may define custom low-level driver interface structure which may be used as FMSTR_SERIAL_DRV. For example:

- **FMSTR_SERIAL_DREG_UART** - demonstrates the low-level interface implemented without the MCUXpresso SDK and using direct access to peripheral registers.

FMSTR_SERIAL_BASE

```
#define FMSTR_SERIAL_BASE [address|symbol]
```

Value Type Optional address value (numeric or symbolic)

Description Specify the base address of the UART, LPUART, USART, or other serial peripheral module to be used for the communication. This value is not defined by default. User application should call `FMSTR_SetSerialBaseAddress()` to select the peripheral module.

FMSTR_COMM_BUFFER_SIZE

```
#define FMSTR_COMM_BUFFER_SIZE [number]
```

Value Type 0 or a value in range 32...255

Description Specify the size of the communication buffer to be allocated by the driver. Default value, which suits all driver features, is used when this option is defined as 0.

FMSTR_COMM_RQUEUE_SIZE

```
#define FMSTR_COMM_RQUEUE_SIZE [number]
```

Value Type Value in range 0...255

Description Specify the size of the FIFO receiver queue used to quickly receive and store characters in the FMSTR_SHORT_INTR interrupt mode.
The default value is 32 B.

FMSTR_SERIAL_SINGLEWIRE

```
#define FMSTR_SERIAL_SINGLEWIRE [0|1]
```

Value Type Boolean 0 or 1.

Description Set to non-zero to enable the "True" single-wire mode which uses a single MCU pin to communicate. The low-level driver enables the pin direction switching when the MCU peripheral supports it.

CAN Bus transport This section describes configuration parameters used when CAN transport is used:

```
#define FMSTR_TRANSPORT FMSTR_CAN
```

FMSTR_CAN_DRV Select what low-level driver interface will be used when implementing the CAN communication.

```
#define FMSTR_CAN_DRV [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing CAN driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/can implementation*):

- **FMSTR_CAN_MCUX_FLEXCAN** - FlexCAN driver
- **FMSTR_CAN_MCUX_MCAN** - MCAN driver
- **FMSTR_CAN_MCUX_MSCAN** - msCAN driver
- **FMSTR_CAN_MCUX_DSCFLEXCAN** - DSC FlexCAN driver
- **FMSTR_CAN_MCUX_DSCMSCAN** - DSC msCAN driver

Other SDKs or BSPs may define the custom low-level driver interface structure which may be used as FMSTR_CAN_DRV.

FMSTR_CAN_BASE

```
#define FMSTR_CAN_BASE [address|symbol]
```

Value Type Optional address value (numeric or symbolic)

Description Specify the base address of the FlexCAN, msCAN, or other CAN peripheral module to be used for the communication. This value is not defined by default. User application should call FMSTR_SetCanBaseAddress() to select the peripheral module.

FMSTR_CAN_CMDID

```
#define FMSTR_CAN_CMDID [number]
```

Value Type CAN identifier (11-bit or 29-bit number)

Description CAN message identifier used for FreeMASTER commands (direction from PC Host tool to target application). When declaring 29-bit identifier, combine the numeric value with FMSTR_CAN_EXTID bit. Default value is 0x7AA.

FMSTR_CAN_RSPID

```
#define FMSTR_CAN_RSPID [number]
```

Value Type CAN identifier (11-bit or 29-bit number)

Description CAN message identifier used for responding messages (direction from target application to PC Host tool). When declaring 29-bit identifier, combine the numeric value with FMSTR_CAN_EXTID bit. Note that both *CMDID* and *RSPID* values may be the same. Default value is 0x7AA.

FMSTR_FLEXCAN_TXMB

```
#define FMSTR_FLEXCAN_TXMB [number]
```

Value Type Number in range of 0..N where N is number of CAN message-buffers supported by HW module.

Description Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for CAN frame transmission. Default value is 0.

FMSTR_FLEXCAN_RXMB

```
#define FMSTR_FLEXCAN_RXMB [number]
```

Value Type Number in range of 0..N where N is number of CAN message-buffers supported by HW module.

Description Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for CAN frame reception. Note that the FreeMASTER driver may also operate with a common message buffer used by both TX and RX directions. Default value is 1.

Network transport This section describes configuration parameters used when Network transport is used:

```
#define FMSTR_TRANSPORT FMSTR_NET
```

FMSTR_NET_DRV Select network interface implementation.

```
#define FMSTR_NET_DRV [identifier]
```

Value Type Identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing NET driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/network implementation*):

- **FMSTR_NET_LWIP_TCP** - TCP communication using lwIP stack
- **FMSTR_NET_LWIP_UDP** - UDP communication using lwIP stack
- **FMSTR_NET_SEGGER_RTT** - Communication using SEGGER J-Link RTT interface

Other SDKs or BSPs may define the custom networking interface which may be used as FMSTR_CAN_DRV.

Add another row below:

FMSTR_NET_PORT

```
#define FMSTR_NET_PORT [number]
```

Value Type TCP or UDP port number (short integer)

Description Specifies the server port number used by TCP or UDP protocols.

FMSTR_NET_BLOCKING_TIMEOUT

```
#define FMSTR_NET_BLOCKING_TIMEOUT [number]
```

Value Type Timeout as number of milliseconds

Description This value specifies a timeout in milliseconds for which the network socket operations may block the execution inside *FMSTR_Poll*. This may be set high (e.g. 250) when a dedicated RTOS task is used to handle FreeMASTER protocol polling. Set to a lower value when the polling task is also responsible for other operations. Set to 0 to attempt to use non-blocking socket operations.

FMSTR_NET_AUTODISCOVERY

```
#define FMSTR_NET_AUTODISCOVERY [0|1]
```

Value Type Boolean 0 or 1.

Description This option enables the FreeMASTER driver to use a separate UDP socket to broadcast auto-discovery messages to network. This helps the FreeMASTER tool to discover the target device address, port and protocol options.

Debugging options

FMSTR_DISABLE

```
#define FMSTR_DISABLE [0|1]
```

Value Type boolean (0 or 1)

Description Define as non-zero to disable all FreeMASTER features, exclude the driver code from build, and compile all its API functions empty. This may be useful to remove FreeMASTER without modifying any application source code. Default value is 0 (false).

FMSTR_DEBUG_TX

```
#define FMSTR_DEBUG_TX [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to enable the driver to periodically transmit test frames out on the selected communication interface (SCI or CAN). With the debug transmission enabled, it is simpler to detect problems in the baudrate or other communication configuration settings.

The test frames are transmitted until the first valid command frame is received from the PC Host tool. The test frame is a valid error status frame, as defined by the protocol format. On the serial line, the test frame consists of three printable characters (+©W) which are easy to capture using the serial terminal tools.

This feature requires the FMSTR_Poll() function to be called periodically. Default value is 0 (false).

FMSTR_APPLICATION_STR

```
#define FMSTR_APPLICATION_STR
```

Value Type String.

Description Name of the application visible in FreeMASTER host application.

Memory access

FMSTR_USE_READMEM

```
#define FMSTR_USE_READMEM [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Memory Read command and enable FreeMASTER to have read access to memory and variables. The access can be further restricted by using a TSA feature.
Default value is 1 (true).

FMSTR_USE_WRITEMEM

```
#define FMSTR_USE_WRITEMEM [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Memory Write command.
The default value is 1 (true).

Oscilloscope options

FMSTR_USE_SCOPE

```
#define FMSTR_USE_SCOPE [number]
```

Value Type Integer number.

Description Number of Oscilloscope instances to be supported. Set to 0 to disable the Oscilloscope feature.
Default value is 0.

FMSTR_MAX_SCOPE_VARS

```
#define FMSTR_MAX_SCOPE_VARS [number]
```

Value Type Integer number larger than 2.

Description Number of variables to be supported by each Oscilloscope instance.
Default value is 8.

Recorder options

FMSTR_USE_RECORDER

```
#define FMSTR_USE_RECORDER [number]
```

Value Type Integer number.

Description Number of Recorder instances to be supported. Set to 0 to disable the Recorder feature.
Default value is 0.

FMSTR_REC_BUFF_SIZE

```
#define FMSTR_REC_BUFF_SIZE [number]
```

Value Type Integer number larger than 2.

Description Defines the size of the memory buffer used by the Recorder instance #0.
Default: not defined, user shall call 'FMSTR_RecorderCreate()" API function to specify this parameter in run time.

FMSTR_REC_TIMEBASE

```
#define FMSTR_REC_TIMEBASE [time specification]
```

Value Type Number (nanoseconds time).

Description Defines the base sampling rate in nanoseconds (sampling speed) Recorder instance #0.

Use one of the following macros:

- FMSTR_REC_BASE_SECONDS(x)
- FMSTR_REC_BASE_MILLISEC(x)
- FMSTR_REC_BASE_MICROSEC(x)
- FMSTR_REC_BASE_NANOSEC(x)

Default: not defined, user shall call 'FMSTR_RecorderCreate()" API function to specify this parameter in run time.

FMSTR_REC_FLOAT_TRIG

```
#define FMSTR_REC_FLOAT_TRIG [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the floating-point triggering. Be aware that floating-point triggering may grow the code size by linking the floating-point standard library.

Default value is 0 (false).

Application Commands options

FMSTR_USE_APPCMD

```
#define FMSTR_USE_APPCMD [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Application Commands feature.
Default value is 0 (false).

FMSTR_APPCMD_BUFF_SIZE

```
#define FMSTR_APPCMD_BUFF_SIZE [size]
```

Value Type Numeric buffer size in range 1..255

Description The size of the Application Command data buffer allocated by the driver. The buffer stores the (optional) parameters of the Application Command which waits to be processed.

FMSTR_MAX_APPCMD_CALLS

```
#define FMSTR_MAX_APPCMD_CALLS [number]
```

Value Type Number in range 0..255

Description The number of different Application Commands that can be assigned a callback handler function using FMSTR_RegisterAppCmdCall(). Default value is 0.

TSA options

FMSTR_USE_TSA

```
#define FMSTR_USE_TSA [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the FreeMASTER TSA feature to be used. With this option enabled, the TSA tables defined in the applications are made available to the FreeMASTER host tool.
Default value is 0 (false).

FMSTR_USE_TSA_SAFETY

```
#define FMSTR_USE_TSA_SAFETY [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the memory access validation in the FreeMASTER driver. With this option, the host tool is not able to access the memory which is not described by at least one TSA descriptor. Also a write access is denied for objects defined as read-only in TSA tables.
Default value is 0 (false).

FMSTR_USE_TSA_INROM

```
#define FMSTR_USE_TSA_INROM [0|1]
```

Value Type Boolean 0 or 1.

Description Declare all TSA descriptors as *const*, which enables the linker to put the data into the flash memory. The actual result depends on linker settings or the linker commands used in the project.
Default value is 0 (false).

FMSTR_USE_TSA_DYNAMIC

```
#define FMSTR_USE_TSA_DYNAMIC [0|1]
```

Value Type Boolean 0 or 1.

Description Enable runtime-defined TSA entries to be added to the TSA table by the FMSTR_SetUpTsaBuff() and FMSTR_TsaAddVar() functions.
Default value is 0 (false).

Pipes options

FMSTR_USE_PIPES

```
#define FMSTR_USE_PIPES [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the FreeMASTER Pipes feature to be used.
Default value is 0 (false).

FMSTR_MAX_PIPES_COUNT

```
#define FMSTR_MAX_PIPES_COUNT [number]
```

Value Type Number in range 1..63.

Description The number of simultaneous pipe connections to support.
The default value is 1.

Driver interrupt modes To implement the communication, the FreeMASTER driver handles the Serial or CAN module's receive and transmit requests. Use the *freemaster_cfg.h* configuration file to select whether the driver processes the communication automatically in the interrupt service routine handler or if it only polls the status of the module (typically during the application idle time).

This section describes each of the interrupt mode in more details.

Completely Interrupt-Driven operation Activated using:

```
#define FMSTR_LONG_INTR 1
```

In this mode, both the communication and the FreeMASTER protocol decoding is done in the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, or other interrupt service routine. Because the protocol execution may be a lengthy task (especially with the TSA-Safety enabled) it is recommended to use this mode only if the interrupt prioritization scheme is possible in the application and the FreeMASTER interrupt is assigned to a lower (the lowest) priority.

In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the FMSTR_SerialIsr or FMSTR_CanIsr functions from that handler.

Mixed Interrupt and Polling Modes Activated using:

```
#define FMSTR_SHORT_INTR 1
```

In this mode, the communication processing time is split between the interrupt routine and the main application loop or task. The raw communication is handled by the *FMSTR_SerialIsr, FMSTR_CanIsr*, or other interrupt service routine, while the protocol decoding and execution is handled by the *FMSTR_Poll* routine. Call *FMSTR_Poll* during the idle time in the application main loop.

The interrupt processing in this mode is relatively fast and deterministic. Upon a serial-receive event, the received character is only placed into a FIFO-like queue and it is not further processed. Upon a CAN receive event, the received frame is stored into a receive buffer. When transmitting, the characters are fetched from the prepared transmit buffer.

In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the *FMSTR_SerialIsr* or *FMSTR_CanIsr* functions from that handler.

When the serial interface is used as the serial communication interface, ensure that the *FMSTR_Poll* function is called at least once per *N* character time periods. *N* is the length of the FreeMASTER FIFO queue (*FMSTR_COMM_RQUEUE_SIZE*) and the character time is the time needed to transmit or receive a single byte over the SCI line.

Completely Poll-driven

```
#define FMSTR_POLL_DRIVEN 1
```

In this mode, both the communication and the FreeMASTER protocol decoding are done in the *FMSTR_Poll* routine. No interrupts are needed and the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, and similar handlers compile to an empty code.

When using this mode, ensure that the *FMSTR_Poll* function is called by the application at least once per the serial "character time" which is the time needed to transmit or receive a single character.

In the latter two modes (*FMSTR_SHORT_INTR* and *FMSTR_POLL_DRIVEN*), the protocol handling takes place in the FMSTR_Poll routine. An application interrupt can occur in the middle of the

Read Memory or Write Memory commands' execution and corrupt the variable being accessed by the FreeMASTER driver. In these two modes, some issues or glitches may occur when using FreeMASTER to visualize or monitor volatile variables modified in interrupt servicing code.

The same issue may appear even in the full interrupt mode (FMSTR_LONG_INTR), if volatile variables are modified in the interrupt code with a priority higher than the priority of the communication interrupt.

Data types Simple portability was one of the main requirements when writing the FreeMASTER driver. This is why the driver code uses the privately-declared data types and the vast majority of the platform-dependent code is separated in the platform-dependent source files. The data types used in the driver API are all defined in the platform-specific header file.

To prevent name conflicts with the symbols used in the application, all data types, macros, and functions have the FMSTR_ prefix. The only global variables used in the driver are the transport and low-level API structures exported from the driver-implementation layer to upper layers. Other than that, all private variables are declared as static and named using the fmstr_ prefix.

Communication interface initialization The FreeMASTER driver does not perform neither the initialization nor the configuration of the peripheral module that it uses to communicate. It is the application startup code responsibility to configure the communication module before the FreeMASTER driver is initialized by the FMSTR_Init call.

When the Serial communication module is used as the FreeMASTER communication interface, configure the UART receive and transmit pins, the serial communication baud rate, parity (no-parity), the character length (eight bits), and the number of stop bits (one) before initializing the FreeMASTER driver. For either the long or the short interrupt modes of the driver (see *Driver interrupt modes*), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to the selected serial peripheral module. Call the FMSTR_SerialIsr function from the application handler.

When a CAN module is used as the FreeMASTER communication interface, configure the CAN receive and transmit pins and the CAN module bit rate before initializing the FreeMASTER driver. For either the long or the short interrupt modes of the driver (see *Driver interrupt modes*), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to the selected CAN peripheral module. Call the FMSTR_CanIsr function from the application handler.

Note: It is not necessary to enable or unmask the serial nor the CAN interrupts before initializing the FreeMASTER driver. The driver enables or disables the interrupts and communication lines, as required during runtime.

FreeMASTER Recorder calls When using the FreeMASTER Recorder in the application (FMSTR_USE_RECORDER > 0), call the FMSTR_RecorderCreate function early after FMSTR_Init to set up each recorder instance to be used in the application. Then call the FMSTR_Recorder function periodically in the code where the data recording should occur. A typical place to call the Recorder routine is at the timer or PWM interrupts, but it can be anywhere else. The example applications provided together with the driver code call the FMSTR_Recorder in the main application loop.

In applications where FMSTR_Recorder is called periodically with a constant period, specify the period in the Recorder configuration structure before calling FMSTR_RecorderCreate. This setting enables the PC Host FreeMASTER tool to display the X-axis of the Recorder graph properly scaled for the time domain.

Driver usage Start using or evaluating FreeMASTER by opening some of the example applications available in the driver setup package.

Follow these steps to enable the basic FreeMASTER connectivity in the application:

- Make sure that all **.c* files of the FreeMASTER driver from the *src/common/platforms/[your_platform]* folder are a part of the project. See *Driver files* for more details.
- Configure the FreeMASTER driver by creating or editing the *freemaster_cfg.h* file and by saving it into the application project directory. See *Driver configuration* for more details.
- Include the *freemaster.h* file into any application source file that makes the FreeMASTER API calls.
- Initialize the Serial or CAN modules. Set the baud rate, parity, and other parameters of the communication. Do not enable the communication interrupts in the interrupt mask registers.
- For the FMSTR_LONG_INTR and FMSTR_SHORT_INTR modes, install the application-specific interrupt routine and call the FMSTR_SerialIsr or FMSTR_CanIsr functions from this handler.
- Call the FMSTR_Init function early on in the application initialization code.
- Call the FMSTR_RecorderCreate functions for each Recorder instance to enable the Recorder feature.
- In the main application loop, call the FMSTR_Poll API function periodically when the application is idle.
- For the FMSTR_SHORT_INTR and FMSTR_LONG_INTR modes, enable the interrupts globally so that the interrupts can be handled by the CPU.

Communication troubleshooting The most common problem that causes communication issues is a wrong baud rate setting or a wrong pin multiplexer setting of the target MCU. When a communication between the PC Host running FreeMASTER and the target MCU cannot be established, try enabling the FMSTR_DEBUG_TX option in the *freemaster_cfg.h* file and call the FMSTR_Poll function periodically in the main application task loop.

With this feature enabled, the FreeMASTER driver periodically transmits a test frame through the Serial or CAN lines. Use a logic analyzer or an oscilloscope to monitor the signals at the communication pins of the CPU device to examine whether the bit rate and signal polarity are configured properly.

Driver API

This section describes the driver Application Programmers' Interface (API) needed to initialize and use the FreeMASTER serial communication driver.

Control API There are three key functions to initialize and use the driver.

FMSTR_Init

Prototype

```
FMSTR_BOOL FMSTR_Init(void);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_protocol.c*

Description This function initializes the internal variables of the FreeMASTER driver and enables the communication interface. This function does not change the configuration of the selected communication module. The hardware module must be initialized before the *FMSTR_Init* function is called.

A call to this function must occur before calling any other FreeMASTER driver API functions.

FMSTR_Poll

Prototype

```
void FMSTR_Poll(void);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_protocol.c*

Description In the poll-driven or short interrupt modes, this function handles the protocol decoding and execution (see *Driver interrupt modes*). In the poll-driven mode, this function also handles the communication interface with the PC. Typically, the *FMSTR_Poll* function is called during the "idle" time in the main application task loop.

To prevent the receive data overflow (loss) on a serial interface, make sure that the FMSTR_Poll function is called at least once per the time calculated as:

*N * Tchar*

where:

- *N* is equal to the length of the receive FIFO queue (configured by the FMSTR_COMM_RQUEUE_SIZE macro). *N* is 1 for the poll-driven mode.
- *Tchar* is the character time, which is the time needed to transmit or receive a single byte over the SCI line.

Note: In the long interrupt mode, this function typically compiles as an empty function and can still be called. It is worthwhile to call this function regardless of the interrupt mode used in the application. This approach enables a convenient switching between the different interrupt modes only by changing the configuration macros in the *freemaster_cfg.h* file.

FMSTR_SerialIsr / FMSTR_CanIsr

Prototype

```
void FMSTR_SerialIsr(void);
void FMSTR_CanIsr(void);
```

- Declaration: *freemaster.h*
- Implementation: *hw-specific low-level driver C file*

Description This function contains the interrupt-processing code of the FreeMASTER driver. In long or short interrupt modes (see *Driver interrupt modes*), this function must be called from the application interrupt service routine registered for the communication interrupt vector. On platforms where the communication module uses multiple interrupt vectors, the application should register a handler for all vectors and call this function at each interrupt.

Note: In a poll-driven mode, this function is compiled as an empty function and does not have to be used.

Recorder API

FMSTR_RecorderCreate

Prototype

```
FMSTR_BOOL FMSTR_RecorderCreate(FMSTR_INDEX recIndex, FMSTR_REC_BUFF* buffCfg);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function registers a recorder instance and enables it to be used by the PC Host tool. Call this function for all recorder instances from 0 to the maximum number defined by the FMSTR_USE_RECORDER configuration option (minus one). An exception to this requirement is the recorder of instance *0* which may be automatically configured by FMSTR_Init when the *freemaster_cfg.h* configuration file defines the *FMSTR_REC_BUFF_SIZE* and *FMSTR_REC_TIMEBASE* options.

For more information, see *Configurable items*.

FMSTR_Recorder

Prototype

```
void FMSTR_Recorder(FMSTR_INDEX recIndex);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function takes a sample of the variables being recorded using the FreeMASTER Recorder instance *recIndex*. If the selected Recorder is not active when the *FMSTR_Recorder* function is being called, the function returns immediately. When the Recorder is active, the values of the variables being recorded are copied into the recorder buffer and the trigger conditions are evaluated.

If a trigger condition is satisfied, the Recorder enters the post-trigger mode, where it counts down the follow-up samples (number of *FMSTR_Recorder* function calls) and de-activates the Recorder when the required post-trigger samples are finished.

The *FMSTR_Recorder* function is typically called in the timer or PWM interrupt service routines. This function can also be called in the application main loop (for testing purposes).

FMSTR_RecorderTrigger

Prototype

```
void FMSTR_RecorderTrigger(FMSTR_INDEX recIndex);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function forces the Recorder trigger condition to happen, which causes the Recorder to be automatically deactivated after the post-trigger samples are sampled. Use this function in the application code for programmatic control over the Recorder triggering. This can be useful when a more complex triggering conditions need to be used.

Fast Recorder API The Fast Recorder feature is not available in the FreeMASTER driver version 3. This feature was heavily dependent on the target platform and it was only available for the 56F8xxxx DSCs.

TSA Tables When the TSA is enabled in the FreeMASTER driver configuration file (by setting the FMSTR_USE_TSA macro to a non-zero value), it defines the so-called TSA tables in the application. This section describes the macros that must to be used to define the TSA tables.

There can be any number of TSA tables spread across the application source files. There must be always exactly one TSA Table List defined, which informs the FreeMASTER driver about the active TSA tables.

When there is at least one TSA table and one TSA Table List defined in the application, the TSA information automatically appears in the FreeMASTER symbols list. The symbols can then be used to create FreeMASTER variables for visualization or control.

TSA table definition The TSA table describes the static or global variables together with their address, size, type, and access-protection information. If the TSA-described variables are of a structure type, the TSA table may also describe this type and provide an access to the individual structure members of the variable.

The TSA table definition begins with the FMSTR_TSA_TABLE_BEGIN macro with a *table_id* identifying the table. The *table_id* shall be a valid C-langiage symbol.

```
FMSTR_TSA_TABLE_BEGIN(table_id)
```

After this opening macro, the TSA descriptors are placed using these macros:

```
/* Adding variable descriptors */
FMSTR_TSA_RW_VAR(name, type)  /* read/write variable entry */
FMSTR_TSA_RO_VAR(name, type)  /* read-only variable entry */

/* Description of complex data types */
FMSTR_TSA_STRUCT(struct_name) /* structure or union type entry */
FMSTR_TSA_MEMBER(struct_name, member_name, type)  /* structure member entry */

/* Memory blocks */
FMSTR_TSA_RW_MEM(name, type, address, size) /* read/write memory block */
FMSTR_TSA_RO_MEM(name, type, address, size) /* read-only memory block */
```

The table is closed using the FMSTR_TSA_TABLE_END macro:

```
FMSTR_TSA_TABLE_END()
```

TSA descriptor parameters The TSA descriptor macros accept these parameters:

- *name* — variable name. The variable must be defined before the TSA descriptor references it.
- *type* — variable or member type. Only one of the pre-defined type constants may be used (see below).
- *struct_name* — structure type name. The type must be defined (typedef) before the TSA descriptor references it.

- *member_name* — structure member name.

Note: The structure member descriptors (FMSTR_TSA_MEMBER) must immediately follow the parent structure descriptor (FMSTR_TSA_STRUCT) in the table.

Note: To write-protect the variables in the FreeMASTER driver (FMSTR_TSA_RO_VAR), enable the TSA-Safety feature in the configuration file.

TSA variable types The table lists *type* identifiers which can be used in TSA descriptors:

Constant	Description
FMSTR_TSA_UINT*n*	Unsigned integer type of size *n* bits (n=8,16,32,64)
FMSTR_TSA_SINT*n*	Signed integer type of size *n* bits (n=8,16,32,64)
FMSTR_TSA_FRAC*n*	Fractional number of size *n* bits (n=16,32,64).
FMSTR_TSA_FRAC_Q(*m,n*)	Signed fractional number in general Q form (m+n+1 total bits)
FMSTR_TSA_FRAC_UQ(*m,n*)	Unsigned fractional number in general UQ form (m+n total bits)
FMSTR_TSA_FLOAT	4-byte standard IEEE floating-point type
FMSTR_TSA_DOUBLE	8-byte standard IEEE floating-point type
FMSTR_TSA_POINTER	Generic pointer type defined (platform-specific 16 or 32 bit)
FMSTR_TSA_USERTYPE(*name*)	Structure or union type declared with FMSTR_TSA_STRUCT record

TSA table list There shall be exactly one TSA Table List in the application. The list contains one entry for each TSA table defined anywhere in the application.

The TSA Table List begins with the FMSTR_TSA_TABLE_LIST_BEGIN macro and continues with the TSA table entries for each table.

```
FMSTR_TSA_TABLE_LIST_BEGIN()

FMSTR_TSA_TABLE(table_id)
FMSTR_TSA_TABLE(table_id2)
FMSTR_TSA_TABLE(table_id3)
...
```

The list is closed with the FMSTR_TSA_TABLE_LIST_END macro:

```
FMSTR_TSA_TABLE_LIST_END()
```

TSA Active Content entries FreeMASTER v2.0 and higher supports TSA Active Content, enabling the TSA tables to describe the memory-mapped files, virtual directories, and URL hyperlinks. FreeMASTER can access such objects similarly to accessing the files and folders on the local hard drive.

With this set of TSA entries, the FreeMASTER pages can be embedded directly into the target MCU flash and accessed by FreeMASTER directly over the communication line. The HTML-coded pages rendered inside the FreeMASTER window can access the TSA Active Content resources using a special URL referencing the *fmstr:* protocol.

This example provides an overview of the supported TSA Active Content entries:

```
FMSTR_TSA_TABLE_BEGIN(files_and_links)

/* Directory entry applies to all subsequent MEMFILE entries */
FMSTR_TSA_DIRECTORY("/text_files")     /* entering a new virtual directory */
```

(continues on next page)

(continued from previous page)

```
/* The readme.txt file will be accessible at the fmstr://text_files/readme.txt URL */
FMSTR_TSA_MEMFILE("readme.txt", readme_txt, sizeof(readme_txt)) /* memory-mapped file */

/* Files can also be specified with a full path so the DIRECTORY entry does not apply */
FMSTR_TSA_MEMFILE("/index.htm", index, sizeof(index))          /* memory-mapped file */
FMSTR_TSA_MEMFILE("/prj/demo.pmp", demo_pmp, sizeof(demo_pmp)) /* memory-mapped file */

/* Hyperlinks can point to a local MEMFILE object or to the Internet */
FMSTR_TSA_HREF("Board's Built-in Welcome Page", "/index.htm")
FMSTR_TSA_HREF("FreeMASTER Home Page", "http://www.nxp.com/freemaster")

/* Project file links simplify opening the projects from any URLs */
FMSTR_TSA_PROJECT("Demonstration Project (embedded)", "/prj/demo.pmp")
FMSTR_TSA_PROJECT("Full Project (online)", "http://mycompany.com/prj/demo.pmp")

FMSTR_TSA_TABLE_END()
```

TSA API

FMSTR_SetUpTsaBuff

Prototype

```
FMSTR_BOOL FMSTR_SetUpTsaBuff(FMSTR_ADDR buffAddr, FMSTR_SIZE buffSize);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_tsa.c*

Arguments

- *buffAddr* [in] - address of the memory buffer for the dynamic TSA table
- *buffSize* [in] - size of the memory buffer which determines the maximum number of TSA entries to be added in the runtime

Description This function must be used to assign the RAM memory buffer to the TSA subsystem when FMSTR_USE_TSA_DYNAMIC is enabled. The memory buffer is then used to store the TSA entries added dynamically to the runtime TSA table using the FMSTR_TsaAddVar function call. The runtime TSA table is processed by the FreeMASTER PC Host tool along with all static tables as soon as the communication port is open.

The size of the memory buffer determines the number of TSA entries that can be added dynamically. Depending on the MCU platform, one TSA entry takes either 8 or 16 bytes.

FMSTR_TsaAddVar

Prototype

```
FMSTR_BOOL FMSTR_TsaAddVar(FMSTR_TSATBL_STRPTR tsaName, FMSTR_TSATBL_STRPTR␣
↪tsaType,
    FMSTR_TSATBL_VOIDPTR varAddr, FMSTR_SIZE32 varSize,
    FMSTR_SIZE flags);
```

- Declaration: *freemaster.h*

- Implementation: *freemaster_tsa.c*

Arguments

- *tsaName* [in] - name of the object
- *tsaType* [in] - name of the object type
- *varAddr* [in] - address of the object
- *varSize* [in] - size of the object
- *flags* [in] - access flags; a combination of these values:
  - *FMSTR_TSA_INFO_RO_VAR* — read-only memory-mapped object (typically a variable)
  - *FMSTR_TSA_INFO_RW_VAR* — read/write memory-mapped object
  - *FMSTR_TSA_INFO_NON_VAR* — other entry, describing structure types, structure members, enumerations, and other types

Description This function can be called only when the dynamic TSA table is enabled by the FMSTR_USE_TSA_DYNAMIC configuration option and when the FMSTR_SetUpTsaBuff function call is made to assign the dynamic TSA table memory. This function adds an entry into the dynamic TSA table. It can be used to register a read-only or read/write memory object or describe an item of the user-defined type.

See *TSA table definition* for more details about the TSA table entries.

Application Commands API

FMSTR_GetAppCmd

Prototype

```
FMSTR_APPCMD_CODE FMSTR_GetAppCmd(void);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Description This function can be used to detect if there is an Application Command waiting to be processed by the application. If no command is pending, this function returns the FMSTR_APPCMDRESULT_NOCMD constant. Otherwise, this function returns the code of the Application Command that must be processed. Use the FMSTR_AppCmdAck call to acknowledge the Application Command after it is processed and to return the appropriate result code to the host.

The FMSTR_GetAppCmd function does not report the commands for which a callback handler function exists. If the FMSTR_GetAppCmd function is called when a callback-registered command is pending (and before it is actually processed by the callback function), this function returns FMSTR_APPCMDRESULT_NOCMD.

FMSTR_GetAppCmdData

Prototype

```
FMSTR_APPCMD_PDATA FMSTR_GetAppCmdData(FMSTR_SIZE* dataLen);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *dataLen* [out] - pointer to the variable that receives the length of the data available in the buffer. It can be NULL when this information is not needed.

Description This function can be used to retrieve the Application Command data when the application determines that an Application Command is pending (see *FMSTR_GetAppCmd*).

There is just a single buffer to hold the Application Command data (the buffer length is FMSTR_APPCMD_BUFF_SIZE bytes). If the data are to be used in the application after the command is processed by the FMSTR_AppCmdAck call, copy the data out to a private buffer.

FMSTR_AppCmdAck

Prototype

```
void FMSTR_AppCmdAck(FMSTR_APPCMD_RESULT resultCode);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *resultCode* [in] - the result code which is to be returned to FreeMASTER

Description This function is used when the Application Command processing finishes in the application. The resultCode passed to this function is returned back to the host and the driver is re-initialized to expect the next Application Command.

After this function is called and before the next Application Command arrives, the return value of the FMSTR_GetAppCmd function is FMSTR_APPCMDRESULT_NOCMD.

FMSTR_AppCmdSetResponseData

Prototype

```
void FMSTR_AppCmdSetResponseData(FMSTR_ADDR resultDataAddr, FMSTR_SIZE resultDataLen);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *resultDataAddr* [in] - pointer to the data buffer that is to be copied to the Application Command data buffer
- *resultDataLen* [in] - length of the data to be copied. It must not exceed the FMSTR_APPCMD_BUFF_SIZE value.

Description This function can be used before the Application Command processing finishes, when there are data to be returned back to the PC.

The response data buffer is copied into the Application Command data buffer, from where it is accessed when the host requires it. Do not use FMSTR_GetAppCmdData and the data buffer after FMSTR_AppCmdSetResponseData is called.

Note: The current version of FreeMASTER does not support the Application Command response data.

FMSTR_RegisterAppCmdCall

Prototype

```
FMSTR_BOOL FMSTR_RegisterAppCmdCall(FMSTR_APPCMD_CODE appCmdCode, FMSTR_
↪PAPPCMDFUNC callbackFunc);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *appCmdCode* [in] - the Application Command code for which the callback is to be registered
- *callbackFunc* [in] - pointer to the callback function that is to be registered. Use NULL to unregister a callback registered previously with this Application Command.

Return value This function returns a non-zero value when the callback function was successfully registered or unregistered. It can return zero when trying to register a callback function for more than FMSTR_MAX_APPCMD_CALLS different Application Commands.

Description This function can be used to register the given function as a callback handler for the Application Command. The Application Command is identified using single-byte code. The callback function is invoked automatically by the FreeMASTER driver when the protocol decoder obtains a request to get the application command result code.

The prototype of the callback function is

```
FMSTR_APPCMD_RESULT HandlerFunction(FMSTR_APPCMD_CODE nAppcmd,
    FMSTR_APPCMD_PDATA pData, FMSTR_SIZE nDataLen);
```

Where:

- *nAppcmd* -Application Command code
- *pData* —points to the Application Command data received (if any)
- *nDataLen* —information about the Application Command data length

The return value of the callback function is used as the Application Command Result Code and returned to FreeMASTER.

Note: The FMSTR_MAX_APPCMD_CALLS configuration macro defines how many different Application Commands may be handled by a callback function. When FMSTR_MAX_APPCMD_CALLS is undefined or defined as zero, the FMSTR_RegisterAppCmdCall function always fails.

Pipes API

FMSTR_PipeOpen

Prototype

```
FMSTR_HPIPE FMSTR_PipeOpen(FMSTR_PIPE_PORT pipePort, FMSTR_PPIPEFUNC pipeCallback,
↪
    FMSTR_ADDR pipeRxBuff, FMSTR_PIPE_SIZE pipeRxSize,
    FMSTR_ADDR pipeTxBuff, FMSTR_PIPE_SIZE pipeTxSize,
    FMSTR_U8 type, const FMSTR_CHAR *name);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipePort* [in] - port number that identifies the pipe for the client
- *pipeCallback* [in] - pointer to the callback function that is called whenever a pipe data status changes
- *pipeRxBuff* [in] - address of the receive memory buffer
- *pipeRxSize* [in] - size of the receive memory buffer
- *pipeTxBuff* [in] - address of the transmit memory buffer
- *pipeTxSize* [in] - size of the transmit memory buffer
- *type* [in] - a combination of FMSTR_PIPE_MODE_xxx and FMSTR_PIPE_SIZE_xxx constants describing primary pipe data format and usage. This type helps FreeMASTER decide how to access the pipe by default. Optional, use 0 when undetermined.
- *name* [in] - user name of the pipe port. This name is visible to the FreeMASTER user when creating the graphical pipe interface.

Description This function initializes a new pipe and makes it ready to accept or send the data to the PC Host client. The receive memory buffer is used to store the received data before they are read out by the FMSTR_PipeRead call. When this buffer gets full, the PC Host client denies the data transmission into this pipe until there is enough free space again. The transmit memory buffer is used to store the data transmitted by the application to the PC Host client using the FMSTR_PipeWrite call. The transmit buffer can get full when the PC Host is disconnected or when it is slow in receiving and reading out the pipe data.

The function returns the pipe handle which must be stored and used in the subsequent calls to manage the pipe object.

The callback function (if specified) is called whenever new data are received through the pipe and available for reading. This callback is also called when the data waiting in the transmit buffer are successfully pushed to the PC Host and the transmit buffer free space increases. The prototype of the callback function provided by the user application must be as follows. The *PipeHandler* name is only a placeholder and must be defined by the application.

```
void PipeHandler(FMSTR_HPIPE pipeHandle);
```

FMSTR_PipeClose

Prototype

```
void FMSTR_PipeClose(FMSTR_HPIPE pipeHandle);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call

Description This function de-initializes the pipe object. No data can be received or sent on the pipe after this call.

FMSTR_PipeWrite

Prototype

```
FMSTR_PIPE_SIZE FMSTR_PipeWrite(FMSTR_HPIPE pipeHandle, FMSTR_ADDR pipeData,
    FMSTR_PIPE_SIZE pipeDataLen, FMSTR_PIPE_SIZE writeGranularity);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call
- *pipeData* [in] - address of the data to be written
- *pipeDataLen* [in] - length of the data to be written
- *writeGranularity* [in] - size of the minimum unit of data which is to be written

Description This function puts the user-specified data into the pipe's transmit memory buffer and schedules it for transmission. This function returns the number of bytes that were successfully written into the buffer. This number may be smaller than the number of the requested bytes if there is not enough free space in the transmit buffer.

The *writeGranularity* argument can be used to split the data into smaller chunks, each of the size given by the *writeGranularity* value. The FMSTR_PipeWrite function writes as many data chunks as possible into the transmit buffer and does not attempt to write an incomplete chunk. This feature can prove to be useful to avoid the intermediate caching when writing an array of integer values or other multi-byte data items. When making the nGranularity value equal to the nLength value, all data are considered as one chunk which is either written successfully as a whole or not at all. The nGranularity value of 0 or 1 disables the data-chunk approach.

FMSTR_PipeRead

Prototype

```
FMSTR_PIPE_SIZE FMSTR_PipeRead(FMSTR_HPIPE pipeHandle, FMSTR_ADDR pipeData,
    FMSTR_PIPE_SIZE pipeDataLen, FMSTR_PIPE_SIZE readGranularity);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call
- *pipeData* [in] - address of the data buffer to be filled with the received data
- *pipeDataLen* [in] - length of the data to be read
- *readGranularity* [in] - size of the minimum unit of data which is to be read

Description This function copies the data received from the pipe from its receive buffer to the user buffer for further processing. The function returns the number of bytes that were successfully copied to the buffer. This number may be smaller than the number of the requested bytes if there is not enough data bytes available in the receive buffer.

The readGranularity argument can be used to copy the data in larger chunks in the same way as described in the FMSTR_PipeWrite function.

API data types This section describes the data types used in the FreeMASTER driver. The information provided here can be useful when modifying or porting the FreeMASTER Communication Driver to new NXP platforms.

Note: The licensing conditions prohibit use of FreeMASTER and the FreeMASTER Communication Driver with non-NXP MPU or MCU products.

Public common types The table below describes the public data types used in the FreeMASTER driver API calls. The data types are declared in the *freemaster.h* header file.

Type name	Description
FMSTR_ADDR	Data type used to hold the memory address. On most platforms, this is normally a C-pointer, but it may also be a pure integer type.
For example, this type is defined as long integer on the 56F8xxx platform where the 24-bit addresses must be supported, but the C-pointer may be only 16 bits wide in some compiler configurations.	
FMSTR_SIZE	Data type used to hold the memory block size.
It is required that this type is unsigned and at least 16 bits wide integer.	
FMSTR_BOOL	Data type used as a general boolean type.
This type is used only in zero/non-zero conditions in the driver code.	
FMSTR_APPCM	Data type used to hold the Application Command code.
Generally, this is an unsigned 8-bit value.	
FMSTR_APPCM	Data type used to create the Application Command data buffer.
Generally, this is an unsigned 8-bit value.	
FMSTR_APPCM	Data type used to hold the Application Command result code.
Generally, this is an unsigned 8-bit value.	

Public TSA types The table describes the TSA-specific public data types. These types are declared in the *freemaster_tsa.h* header file, which is included in the user application indirectly by the *freemaster.h* file.

FMSTR_TSA_TI	Data type used to hold a descriptor index in the TSA table or a table index in the list of TSA tables.
By default, this is defined as FMSTR_SIZE.	
FMSTR_TSA_TS	Data type used to hold a memory block size, as used in the TSA descriptors.
By default, this is defined as FMSTR_SIZE.	

Public Pipes types The table describes the data types used by the FreeMASTER Pipes API:

FMSTR_HPIPE	Pipe handle that identifies the open-pipe object.
Generally, this is a pointer to a void type.	
FMSTR_PIPE_P	Integer type required to hold at least 7 bits of data.
Generally, this is an unsigned 8-bit or 16-bit type.	
FMSTR_PIPE_SI	Integer type required to hold at least 16 bits of data.
This is used to store the data buffer sizes.	
FMSTR_PPIPEF	Pointer to the pipe handler function.
See *FMSTR_PipeOpen* for more details.	

Internal types The table describes the data types used internally by the FreeMASTER driver. The data types are declared in the platform-specific header file and they are not available in the application code.

FMSTR_U8	The smallest memory entity.
On the vast majority of platforms, this is an unsigned 8-bit integer.	
On the 56F8xx DSP platform, this is defined as an unsigned 16-bit integer.	
FMSTR_U16	Unsigned 16-bit integer.
FMSTR_U32	Unsigned 32-bit integer.
FMSTR_S8	Signed 8-bit integer.
FMSTR_S16	Signed 16-bit integer.
FMSTR_S32	Signed 32-bit integer.
FMSTR_FLOAT	4-byte standard IEEE floating-point type.
FMSTR_FLAGS	Data type forming a union with a structure of flag bit-fields.
FMSTR_SIZE8	Data type holding a general size value, at least 8 bits wide.
FMSTR_INDEX	General for-loop index. Must be signed, at least 16 bits wide.
FMSTR_BCHR	A single character in the communication buffer.
Typically, this is an 8-bit unsigned integer, except for the DSP platforms where it is a 16-bit integer.	
FMSTR_BPTR	A pointer to the communication buffer (an array of FMSTR_BCHR).

Document references

Links

- This document online: https://mcuxpresso.nxp.com/mcuxsdk/latest/html/middleware/freemaster/doc/index.html

- FreeMASTER tool home: www.nxp.com/freemaster
- FreeMASTER community area: community.nxp.com/community/freemaster
- FreeMASTER GitHub code repo: https://github.com/nxp-mcuxpresso/mcux-freemaster
- MCUXpresso SDK home: www.nxp.com/mcuxpresso
- MCUXpresso SDK builder: mcuxpresso.nxp.com/en

Documents

- *FreeMASTER Usage Serial Driver Implementation* (document AN4752)
- *Integrating FreeMASTER Time Debugging Tool With CodeWarrior For Microcontrollers v10.X Project* (document AN4771)
- *Flash Driver Library For MC56F847xx And MC56F827xx DSC Family* (document AN4860)

Revision history This Table summarizes the changes done to this document since the initial release.

Revision	Date	Description
1.0	03/2006	Limited initial release
2.0	09/2007	Updated for FreeMASTER version. New Freescale document template used.
2.1	12/2007	Added description of the new Fast Recorder feature and its API.
2.2	04/2010	Added support for MPC56xx platform, Added new API for use CAN interface.
2.3	04/2011	Added support for Kxx Kinetis platform and MQX operating system.
2.4	06/2011	Serial driver update, adds support for USB CDC interface.
2.5	08/2011	Added Packet Driven BDM interface.
2.7	12/2013	Added FLEXCAN32 interface, byte access and isr callback configuration option.
2.8	06/2014	Removed obsolete license text, see the software package content for up-to-date license.
2.9	03/2015	Update for driver version 1.8.2 and 1.9: FreeMASTER Pipes, TSA Active Content, LIN Transport Layer support, DEBUG-TX communication troubleshooting, Kinetis SDK support.
3.0	08/2016	Update for driver version 2.0: Added support for MPC56xx, MPC57xx, KEAxx and S32Kxx platforms. New NXP document template as well as new license agreement used. added MCAN interface. Folders structure at the installation destination was rearranged.
4.0	04/2019	Update for driver released as part of FreeMASTER v3.0 and MCUXpresso SDK 2.6. Updated to match new V4 serial communication protocol and new configuration options. This version of the document removes substantial portion of outdated information related to S08, S12, ColdFire, Power and other legacy platforms.
4.1	04/2020	Minor update for FreeMASTER driver included in MCUXpresso SDK 2.8.
4.2	09/2020	Added example applications description and information about the MCUXpresso Config Tools. Fixed the pipe-related API description.
4.3	10/2024	Added description of Network and Segger J-Link RTT interface configuration. Accompanying the MCUXpresso SDK version 24.12.00.
4.4	04/2025	Added Zephyr-specific information. Accompanying the MCUXpresso SDK version 25.06.00.

3.2 MultiCore

3.2.1 Multicore SDK

Multicore Software Development Kit (MCSDK) is a Software Development Kit that provides comprehensive software support for NXP dual/multicore devices. The MCSDK is combined with the MCUXpresso SDK to make the software framework for easy development of multicore applications.

Multicore SDK (MCSDK) Release Notes

Overview These are the release notes for the NXP Multicore Software Development Kit (MCSDK) version 25.12.00.
This software package contains components for efficient work with multicore devices as well as for the
multiprocessor communication.

What is new

- eRPC CHANGELOG
- RPMsg-Lite CHANGELOG
- MCMgr CHANGELOG
- Supported evaluation boards (multicore examples):
  - LPCXpresso55S69
  - FRDM-K32L3A6
  - MIMXRT1170-EVKB
  - MIMXRT1160-EVK
  - MIMXRT1180-EVK
  - MCX-N5XX-EVK
  - MCX-N9XX-EVK
  - FRDM-MCXN947
  - MIMXRT700-EVK
  - KW47-EVK
  - KW47-LOC
  - FRDM-MCXW72
  - MCX-W72-EVK
  - FRDM-IMXRT1186
- Supported evaluation boards (multiprocessor examples):
  - LPCXpresso55S36
  - FRDM-K22F
  - FRDM-K32L2B
  - MIMXRT685-EVK
  - MIMXRT1170-EVKB
  - MIMXRT1180
  - FRDM-MCXN236
  - FRDM-MCXC242
  - FRDM-MCXC444
  - MCX-N9XX-EVK
  - FRDM-MCXN947
  - MIMXRT700-EVK
  - FRDM-IMXRT1186

Development tools The Multicore SDK (MCSDK) was compiled and tested with development tools referred in: Development tools

Release contents This table describes the release contents. Not all MCUXpresso SDK packages contain the whole set of these components.

Deliverable	Location
Multicore SDK location <MCSDK_dir>	<MCUXpressoSDK_install_dir>/middleware/multicore/
Documentation	<MCSDK_dir>/mcuxsdk-doc/
Embedded Remote Procedure Call component	<MCSDK_dir>/erpc/
Multicore Manager component	<MCSDK_dir>/mcmgr/
RPMsg-Lite	<MCSDK_dir>/rpmsg_lite/
Multicore demo applications	<MCUXpressoSDK_install_dir>/examples/multicore_examples/
Multiprocessor demo applications	<MCUXpressoSDK_install_dir>/examples/multiprocessor_examples/

Multicore SDK release overview Together, the Multicore SDK (MCSDK) and the MCUXpresso SDK (SDK) form a framework for the development of software for NXP multicore devices. The MCSDK release consists of the following elementary software components for multicore:

- Embedded Remote Procedure Call (eRPC)
- Multicore Manager (MCMGR) - included just in SDK for multicore devices
- Remote Processor Messaging - Lite (RPMsg-Lite) - included just in SDK for multicore devices

The MCSDK is also accompanied with documentation and several multicore and multiprocessor demo applications.

Demo applications The multicore demo applications demonstrate the usage of the MCSDK software components on supported multicore development boards.
The following multicore demo applications are located together with other MCUXpresso SDK examples in
the <MCUXpressoSDK_install_dir>/examples/multicore_examples subdirectories.

- erpc_matrix_multiply_mu
- erpc_matrix_multiply_mu_rtos
- erpc_matrix_multiply_rpmsg
- erpc_matrix_multiply_rpmsg_rtos
- erpc_two_way_rpc_rpmsg_rtos
- freertos_message_buffers
- hello_world
- multicore_manager
- rpmsg_lite_pingpong
- rpmsg_lite_pingpong_rtos
- rpmsg_lite_pingpong_dsp
- rpmsg_lite_pingpong_tzm

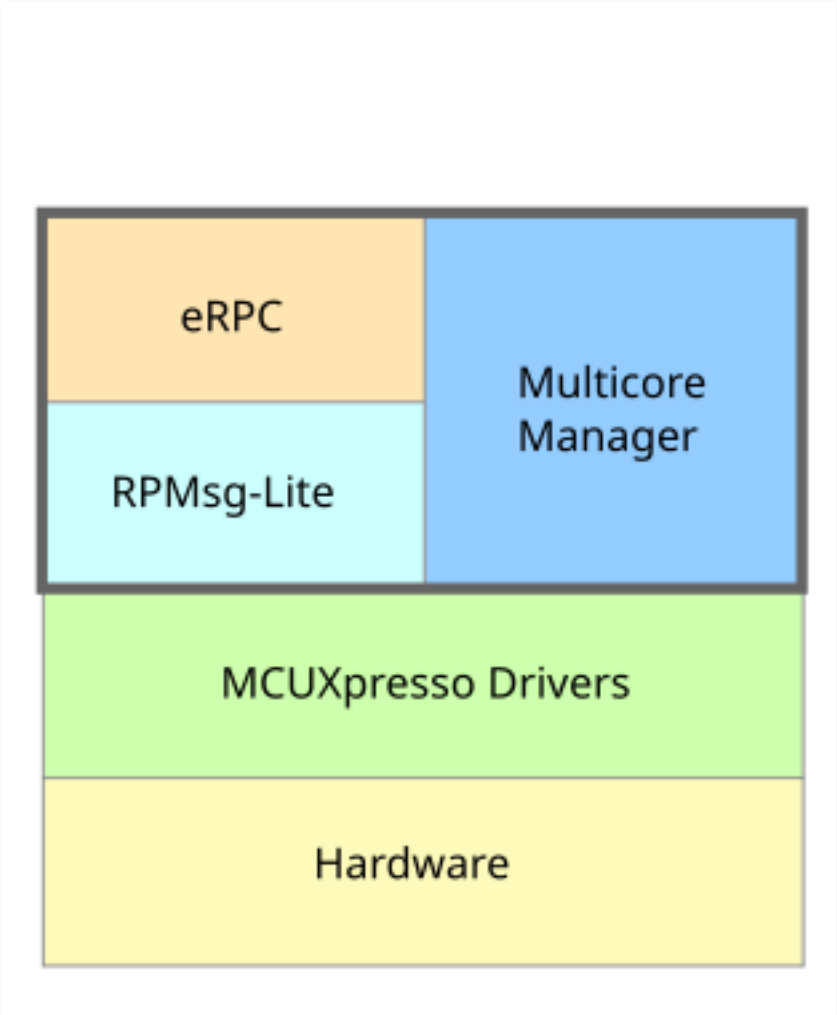
The eRPC multicore component can be leveraged for inter-processor communication and remote procedure calls between SoCs / development boards.
The following multiprocessor demo applications are located together with other MCUXpresso SDK examples in
the <MCUXpressoSDK_install_dir>/examples/multiprocessor_examples subdirectories.

- erpc_client_matrix_multiply_spi
- erpc_server_matrix_multiply_spi
- erpc_client_matrix_multiply_uart
- erpc_server_matrix_multiply_uart
- erpc_server_dac_adc
- erpc_remote_control

Getting Started with Multicore SDK (MCSDK)

Overview Multicore Software Development Kit (MCSDK) is a Software Development Kit that provides comprehensive software support for NXP dual/multicore devices. The MCSDK is combined with the MCUXpresso SDK to make the software framework for easy development of multicore applications.

The following figure highlights the layers and main software components of the MCSDK.

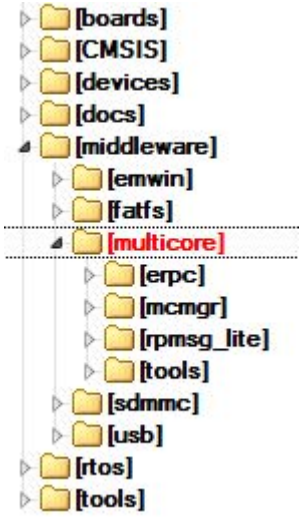


All the MCSDK-related files are located in <MCUXpressoSDK_install_dir>/middleware/multicore folder.

For supported toolchain versions, see the *Multicore SDK v25.12.00 Release Notes* (document MCSDKRN). For the latest version of this and other MCSDK documents, visit www.nxp.com.

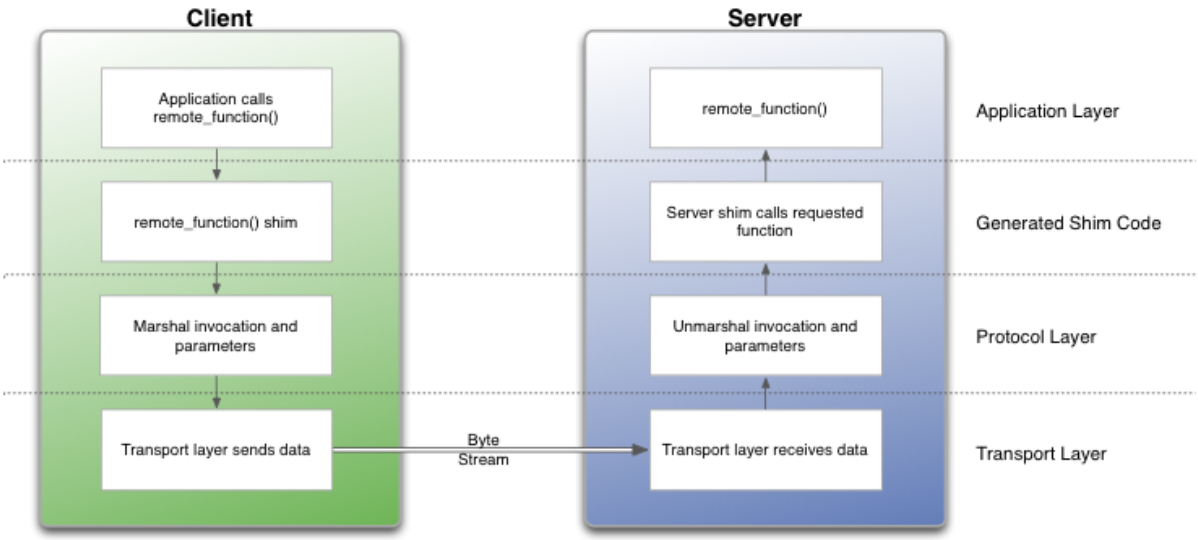
Multicore SDK (MCSDK) components The MCSDK consists of the following software components:

- **Embedded Remote Procedure Call (eRPC):** This component is a combination of a library and code generator tool that implements a transparent function call interface to remote services (running on a different core).
- **Multicore Manager (MCMGR):** This library maintains information about all cores and starts up secondary/auxiliary cores.
- **Remote Processor Messaging - Lite (RPMsg-Lite):** Inter-Processor Communication library.



Embedded Remote Procedure Call (eRPC) The Embedded Remote Procedure Call (eRPC) is the RPC system created by NXP. The RPC is a mechanism used to invoke a software routine on a remote system via a simple local function call.

When a remote function is called by the client, the function's parameters and an identifier for the called routine are marshaled (or serialized) into a stream of bytes. This byte stream is transported to the server through a communications channel (IPC, TPC/IP, UART, and so on). The server unmarshaled the parameters, determines which function was invoked, and calls it. If the function returns a value, it is marshaled and sent back to the client.



RPC implementations typically use a combination of a tool (erpcgen) and IDL (interface definition language) file to generate source code to handle the details of marshaling a function's parameters and building the data stream.

Main eRPC features:

- Scalable from BareMetal to Linux OS - configurable memory and threading policies.
- Focus on embedded systems - intrinsic support for C, modular, and lightweight implementation.
- Abstracted transport interface - RPMsg is the primary transport for multicore, UART, or SPI-based solutions can be used for multichip.

The eRPC library is located in the <MCUXpressoSDK_install_dir>/middleware/multicore/erpc folder. For detailed information about the eRPC, see the documentation available in the <MCUXpressoSDK_install_dir>/middleware/multicore/erpc/doc folder.

Multicore Manager (MCMGR) The Multicore Manager (MCMGR) software library provides a number of services for multicore systems.

The main MCMGR features:

- Maintains information about all cores in system.
- Secondary/auxiliary cores startup and shutdown.
- Remote core monitoring and event handling.

The MCMGR library is located in the <MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr folder. For detailed information about the MCMGR library, see the documentation available in the <MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/doc folder.

Remote Processor Messaging Lite (RPMsg-Lite) RPMsg-Lite is a lightweight implementation of the RPMsg protocol. The RPMsg protocol defines a standardized binary interface used to communicate between multiple cores in a heterogeneous multicore system. Compared to the legacy OpenAMP implementation, RPMsg-Lite offers a code size reduction, API simplification, and improved modularity.

The main RPMsg protocol features:

- Shared memory interprocessor communication.
- Virtio-based messaging bus.
- Application-defined messages sent between endpoints.

- Portable to different environments/platforms.
- Available in upstream Linux OS.

The RPMsg-Lite library is located in the <MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg-lite folder. For detailed information about the RPMsg-Lite, see the RPMsg-Lite User's Guide located in the <MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/doc folder.

MCSDK demo applications Multicore and multiprocessor example applications are stored together with other MCUXpresso SDK examples, in the dedicated multicore subfolder.

Location	Folder
Multicore example projects	<MCUXpressoSDK_install_dir>/examples/multicore_examples/<application_name>/
Multiprocessor example projects	<MCUXpressoSDK_install_dir>/examples/multiprocessor_examples/<application_name>/

See the *Getting Started with MCUXpresso SDK* (document MCUXSDKGSUG) and *Getting Started with MCUXpresso SDK for XXX Derivatives* documents for more information about the MCUXpresso SDK example folder structure and the location of individual files that form the example application projects. These documents also contain information about building, running, and debugging multicore demo applications in individual supported IDEs. Each example application also contains a readme file that describes the operation of the example and required setup steps.

Inter-Processor Communication (IPC) levels The MCSDK provides several mechanisms for Inter-Processor Communication (IPC). Particular ways and levels of IPC are described in this chapter.

IPC using low-level drivers

The NXP multicore SoCs are equipped with peripheral modules dedicated for data exchange between individual cores. They deal with the Mailbox peripheral for LPC parts and the Messaging Unit (MU) peripheral for Kinetis and i.MX parts. The common attribute of both modules is the ability to provide a means of IPC, allowing multiple CPUs to share resources and communicate with each other in a simple manner.

The most lightweight method of IPC uses the MCUXpresso SDK low-level drivers for these peripherals. Using the Mailbox/MU driver API functions, it is possible to pass a value from core to core via the dedicated registers (could be a scalar or a pointer to shared memory) and also to trigger inter-core interrupts for notifications.

For details about individual driver API functions, see the MCUXpresso SDK API Reference Manual of the specific multicore device. The MCUXpresso SDK is accompanied with the RPMsg-Lite documentation that shows how to use this API in multicore applications.

Messaging mechanism

On top of Mailbox/MU drivers, a messaging system can be implemented, allowing messages to send between multiple endpoints created on each of the CPUs. The RPMsg-Lite library of the MCSDK provides this ability and serves as the preferred MCUXpresso SDK messaging library. It implements ring buffers in shared memory for messages exchange without the need of a locking mechanism.

The RPMsg-Lite provides the abstraction layer and can be easily ported to different multicore platforms and environments (Operating Systems). The advantages of such a messaging system are ease of use (there is no need to study behavior of the used underlying hardware) and smooth application code portability between platforms due to unified messaging API.

However, this costs several kB of code and data memory. The MCUXpresso SDK is accompanied by the RPMsg-Lite documentation and several multicore examples. You can also obtain the latest RPMsg-Lite code from the GitHub account github.com/nxp-mcuxpresso/rpmsg-lite.

Remote procedure calls

To facilitate the IPC even more and to allow the remote functions invocation, the remote procedure call mechanism can be implemented. The eRPC of the MCSDK serves for these purposes and allows the ability to invoke a software routine on a remote system via a simple local function call. Utilizing different transport layers, it is possible to communicate between individual cores of multicore SoCs (via RPMsg-Lite) or between separate processors (via SPI, UART, or TCP/IP). The eRPC is mostly applicable to the MPU parts with enough of memory resources like i.MX parts.

The eRPC library allows you to export existing C functions without having to change their prototypes (in most cases). It is accompanied by the code generator tool that generates the shim code for serialization and invocation based on the IDL file with definitions of data types and remote interfaces (API).

If the communicating peer is running as a Linux OS user-space application, the generated code can be either in C/C++ or Python.

Using the eRPC simplifies the access to services implemented on individual cores. This way, the following types of applications running on dedicated cores can be easily interfaced:

- Communication stacks (USB, Thread, Bluetooth Low Energy, Zigbee)
- Sensor aggregation/fusion applications
- Encryption algorithms
- Virtual peripherals

The eRPC is publicly available from the following GitHub account: github.com/EmbeddedRPC/erpc. Also, the MCUXpresso SDK is accompanied by the eRPC code and several multicore and multiprocessor eRPC examples.

The mentioned IPC levels demonstrate the scalability of the Multicore SDK library. Based on application needs, different IPC techniques can be used. It depends on the complexity, required speed, memory resources, system design, and so on. The MCSDK brings users the possibility for quick and easy development of multicore and multiprocessor applications.

Changelog Multicore SDK

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

[25.12.00]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.14.0
  - eRPC generator (erpcgen) v1.14.0
  - Multicore Manager (MCMgr) v5.0.2
  - RPMsg-Lite v5.3.0

[25.09.00]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.14.0

  - eRPC generator (erpcgen) v1.14.0
  - Multicore Manager (MCMgr) v5.0.1
  - RPMsg-Lite v5.2.1

[25.06.00]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.14.0
  - eRPC generator (erpcgen) v1.14.0
  - Multicore Manager (MCMgr) v5.0.0
  - RPMsg-Lite v5.2.0

[25.03.00]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.13.0
  - eRPC generator (erpcgen) v1.13.0
  - Multicore Manager (MCMgr) v4.1.7
  - RPMsg-Lite v5.1.4

[24.12.00]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.13.0
  - eRPC generator (erpcgen) v1.13.0
  - Multicore Manager (MCMgr) v4.1.6
  - RPMsg-Lite v5.1.3

[2.16.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.13.0
  - eRPC generator (erpcgen) v1.13.0
  - Multicore Manager (MCMgr) v4.1.5
  - RPMsg-Lite v5.1.2

[2.15.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.12.0
  - eRPC generator (erpcgen) v1.12.0
  - Multicore Manager (MCMgr) v4.1.5
  - RPMsg-Lite v5.1.1

[2.14.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.11.0
  - eRPC generator (erpcgen) v1.11.0
  - Multicore Manager (MCMgr) v4.1.4
  - RPMsg-Lite v5.1.0

[2.13.0_imxrt1180a0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.10.0
  - eRPC generator (erpcgen) v1.10.0
  - Multicore Manager (MCMgr) v4.1.3
  - RPMsg-Lite v5.0.0

[2.13.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.10.0
  - eRPC generator (erpcgen) v1.10.0
  - Multicore Manager (MCMgr) v4.1.3
  - RPMsg-Lite v5.0.0

[2.12.0_imx93]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.9.1
  - eRPC generator (erpcgen) v1.9.1
  - Multicore Manager (MCMgr) v4.1.2
  - RPMsg-Lite v4.0.1

[2.12.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.9.1
  - eRPC generator (erpcgen) v1.9.1
  - Multicore Manager (MCMgr) v4.1.2
  - RPMsg-Lite v4.0.0

[2.11.1]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.9.0
  - eRPC generator (erpcgen) v1.9.0
  - Multicore Manager (MCMgr) v4.1.1
  - RPMsg-Lite v3.2.1

[2.11.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.9.0
  - eRPC generator (erpcgen) v1.9.0
  - Multicore Manager (MCMgr) v4.1.1
  - RPMsg-Lite v3.2.0

[2.10.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.8.1
  - eRPC generator (erpcgen) v1.8.1
  - Multicore Manager (MCMgr) v4.1.1
  - RPMsg-Lite v3.1.2

[2.9.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.8.0
  - eRPC generator (erpcgen) v1.8.0
  - Multicore Manager (MCMgr) v4.1.1
  - RPMsg-Lite v3.1.1

[2.8.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.7.4
  - eRPC generator (erpcgen) v1.7.4
  - Multicore Manager (MCMgr) v4.1.0
  - RPMsg-Lite v3.1.0

[2.7.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.7.3
  - eRPC generator (erpcgen) v1.7.3
  - Multicore Manager (MCMgr) v4.1.0
  - RPMsg-Lite v3.0.0

[2.6.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.7.2
  - eRPC generator (erpcgen) v1.7.2
  - Multicore Manager (MCMgr) v4.0.3
  - RPMsg-Lite v2.2.0

[2.5.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.7.1
  - eRPC generator (erpcgen) v1.7.1
  - Multicore Manager (MCMgr) v4.0.2
  - RPMsg-Lite v2.0.2

[2.4.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.7.0
  - eRPC generator (erpcgen) v1.7.0
  - Multicore Manager (MCMgr) v4.0.1
  - RPMsg-Lite v2.0.1

[2.3.1]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.6.0
  - eRPC generator (erpcgen) v1.6.0
  - Multicore Manager (MCMgr) v4.0.0
  - RPMsg-Lite v1.2.0

[2.3.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.5.0
  - eRPC generator (erpcgen) v1.5.0
  - Multicore Manager (MCMgr) v3.0.0
  - RPMsg-Lite v1.2.0

[2.2.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.4.0
  - eRPC generator (erpcgen) v1.4.0
  - Multicore Manager (MCMgr) v2.0.1
  - RPMsg-Lite v1.1.0

[2.1.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.3.0
  - eRPC generator (erpcgen) v1.3.0

[2.0.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.2.0
  - eRPC generator (erpcgen) v1.2.0
  - Multicore Manager (MCMgr) v2.0.0
  - RPMsg-Lite v1.0.0

[1.1.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.1.0
  - Multicore Manager (MCMgr) v1.1.0
  - Open-AMP / RPMsg based on SHA1 ID 44b5f3c0a6458f3cf80 rev01

[1.0.0]

- Multicore SDK component versions:
  - embedded Remote Procedure Call (eRPC) v1.0.0
  - Multicore Manager (MCMgr) v1.0.0
  - Open-AMP / RPMsg based on SHA1 ID 44b5f3c0a6458f3cf80 rev00

Multicore SDK Components

RPMSG-Lite

MCUXpresso SDK : mcuxsdk-middleware-rpmsg-lite

Overview This repository is for MCUXpresso SDK RPMSG-Lite middleware delivery and it contains RPMSG-Lite component officially provided in NXP MCUXpresso SDK. This repository is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk for the complete delivery of MCUXpresso SDK to be able to build and run RPMSG-Lite examples that are based on mcux-sdk-middleware-rpmsg-lite component.

Documentation Overall details can be reviewed here: MCUXpresso SDK Online Documentation

Visit RPMSG-Lite - Documentation to review details on the contents in this sub-repo.

For Further API documentation, please look at doxygen documentation

Setup Instructions on how to install the MCUXpresso SDK provided from GitHub via west manifest Getting Started with SDK - Detailed Installation Instructions

Contribution We welcome and encourage the community to submit patches directly to the rpmsg-lite project placed on github. Contributing can be managed via pull-requests. Before a pull-request is created the code should be tested and properly formatted.

RPMSG-Lite This documentation describes the RPMsg-Lite component, which is a lightweight implementation of the Remote Processor Messaging (RPMsg) protocol. The RPMsg protocol defines a standardized binary interface used to communicate between multiple cores in a heterogeneous multicore system.

Compared to the RPMsg implementation of the Open Asymmetric Multi Processing (OpenAMP) framework (https://github.com/OpenAMP/open-amp), the RPMsg-Lite offers a code size reduction, API simplification, and improved modularity. On smaller Cortex-M0+ based systems, it is recommended to use RPMsg-Lite.

The RPMsg-Lite is an open-source component developed by NXP Semiconductors and released under the BSD-compatible license.

For overview please read RPMSG-Lite VirtIO Overview.

For RPMSG-Lite Design Considerations please read RPMSG-Lite Design Considerations.

Motivation to create RPMsg-Lite There are multiple reasons why RPMsg-Lite was developed. One reason is the need for the small footprint of the RPMsg protocol-compatible communication component, another reason is the simplification of extensive API of OpenAMP RPMsg implementation.

RPMsg protocol was not documented, and its only definition was given by the Linux Kernel and legacy OpenAMP implementations. This has changed with [1] which is a standardization protocol allowing multiple different implementations to coexist and still be mutually compatible.

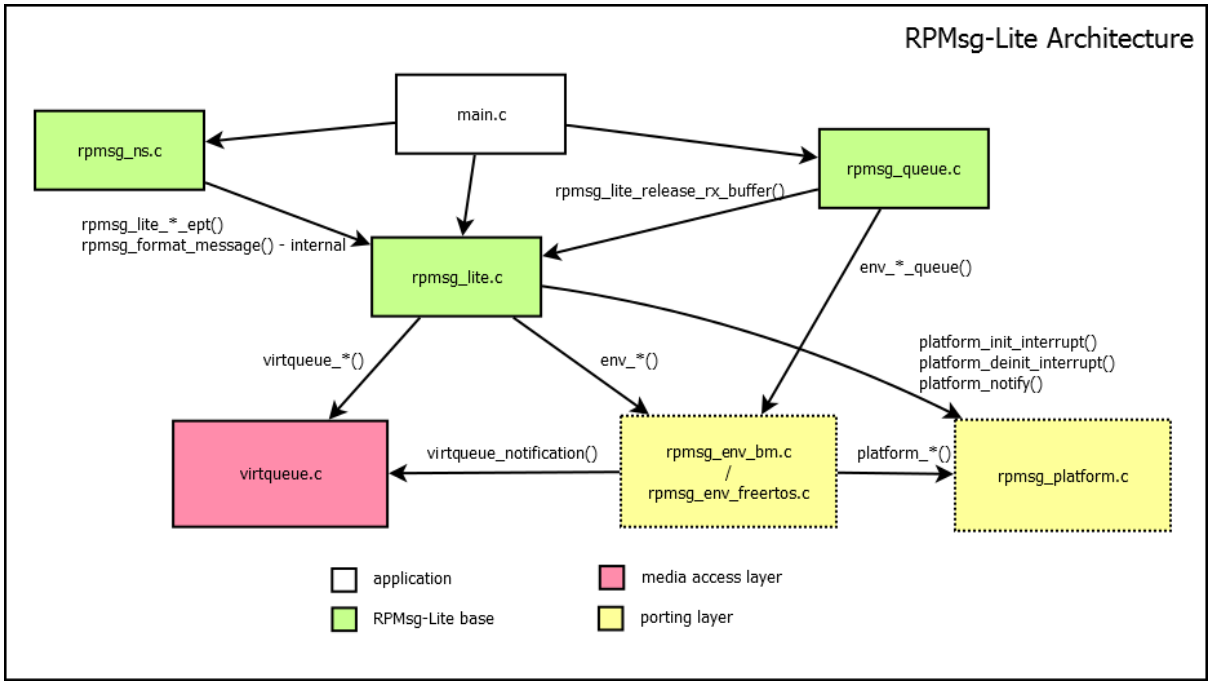
Small MCU-based systems often do not implement dynamic memory allocation. The creation of static API in RPMsg-Lite enables another reduction of resource usage. Not only does the dynamic allocation adds another 5 KB of code size, but also communication is slower and less deterministic, which is a property introduced by dynamic memory. The following table shows some rough comparison data between the OpenAMP RPMsg implementation and new RPMsg-Lite implementation:

Component / Configuration	Flash [B]	RAM [B]
OpenAMP RPMsg / Release (reference)	5547	456 + dynamic
RPMsg-Lite / Dynamic API, Release	3462	56 + dynamic
Relative Difference [%]	~62.4%	~12.3%
RPMsg-Lite / Static API (no malloc), Release	2926	352
Relative Difference [%]	~52.7%	~77.2%

Implementation The implementation of RPMsg-Lite can be divided into three subcomponents, from which two are optional. The core component is situated in rpmsg_lite.c. Two optional components are used to implement a blocking receive API (in rpmsg_queue.c) and dynamic "named" endpoint creation and deletion announcement service (in rpmsg_ns.c).

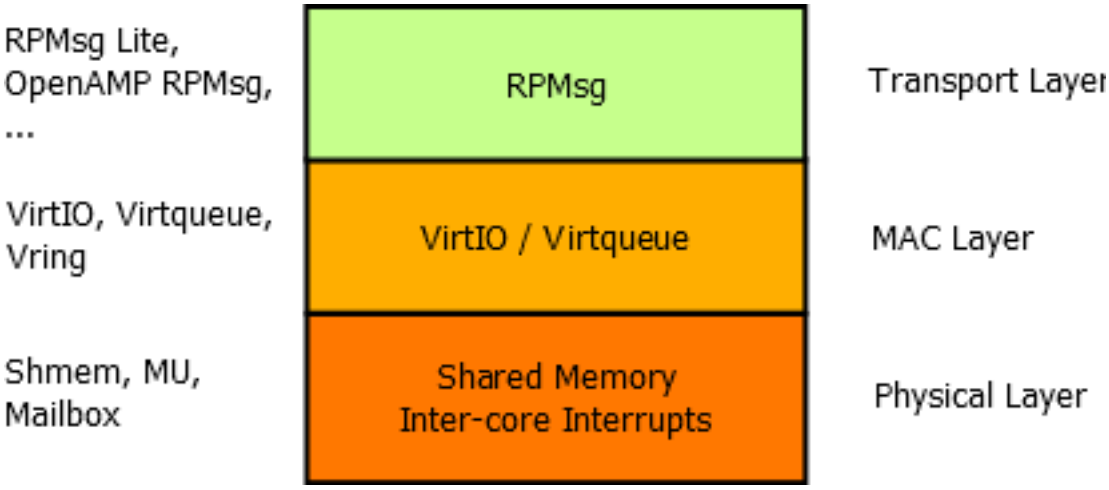
The actual "media access" layer is implemented in virtqueue.c, which is one of the few files shared with the OpenAMP implementation. This layer mainly defines the shared memory model, and internally defines used components such as vring or virtqueue.

The porting layer is split into two sub-layers: the environment layer and the platform layer. The first sublayer is to be implemented separately for each environment. (The bare metal environment already exists and is implemented in rpmsg_env_bm.c, and the FreeRTOS environment is implemented in rpmsg_env_freertos.c etc.) Only the source file, which matches the used environment, is included in the target application project. The second sublayer is implemented in rpmsg_platform.c and defines low-level functions for interrupt enabling, disabling, and triggering mainly. The situation is described in the following figure:



RPMsg-Lite core sub-component This subcomponent implements a blocking send API and callback-based receive API. The RPMsg protocol is part of the transport layer. This is realized by using so-called endpoints. Each endpoint can be assigned a different receive callback function.

However, it is important to notice that the callback is executed in an interrupt environment in current design. Therefore, certain actions like memory allocation are discouraged to execute in the callback. The following figure shows the role of RPMsg in an ISO/OSI-like layered model:



Queue sub-component (optional) This subcomponent is optional and requires implementation of the env_*_queue() functions in the environment porting layer. It uses a blocking receive API, which is common in RTOS-environments. It supports both copy and nocopy blocking receive functions.

Name Service sub-component (optional) This subcomponent is a minimum implementation of the name service which is present in the Linux Kernel implementation of RPMsg. It allows the communicating node both to send announcements about "named" endpoint (in other words, channel) creation or deletion and to receive these announcement taking any user-defined action in an application callback. The endpoint address used to receive name service announcements is arbitrarily fixed to be 53 (0x35).

Usage The application should put the /rpmsg_lite/lib/include directory to the include path and in the application, include either the rpmsg_lite.h header file, or optionally also include the rpmsg_queue.h and/or rpmsg_ns.h files. Both porting sublayers should be provided for you by NXP, but if you plan to use your own RTOS, all you need to do is to implement your own environment layer (in other words, rpmsg_env_myrtos.c) and to include it in the project build.

The initialization of the stack is done by calling the rpmsg_lite_master_init() on the master side and the rpmsg_lite_remote_init() on the remote side. This initialization function must be called prior to any RPMsg-Lite API call. After the init, it is wise to create a communication endpoint, otherwise communication is not possible. This can be done by calling the rpmsg_lite_create_ept() function. It optionally accepts a last argument, where an internal context of the endpoint is created, just in case the RL_USE_STATIC_API option is set to 1. If not, the stack internally calls env_alloc() to allocate dynamic memory for it. In case a callback-based receiving is to be used, an ISR-callback is registered to each new endpoint with user-defined callback data pointer. If a blocking receive is desired (in case of RTOS environment), the rpmsg_queue_create() function must be called before calling rpmsg_lite_create_ept(). The queue handle is passed to the endpoint creation function as a callback data argument and the callback function is set to rpmsg_queue_rx_cb(). Then, it is possible to use rpmsg_queue_receive() function to listen on a queue object for incoming messages. The rpmsg_lite_send() function is used to send messages to the other side.

The RPMsg-Lite also implements no-copy mechanisms for both sending and receiving operations. These methods require specifics that have to be considered when used in an application.

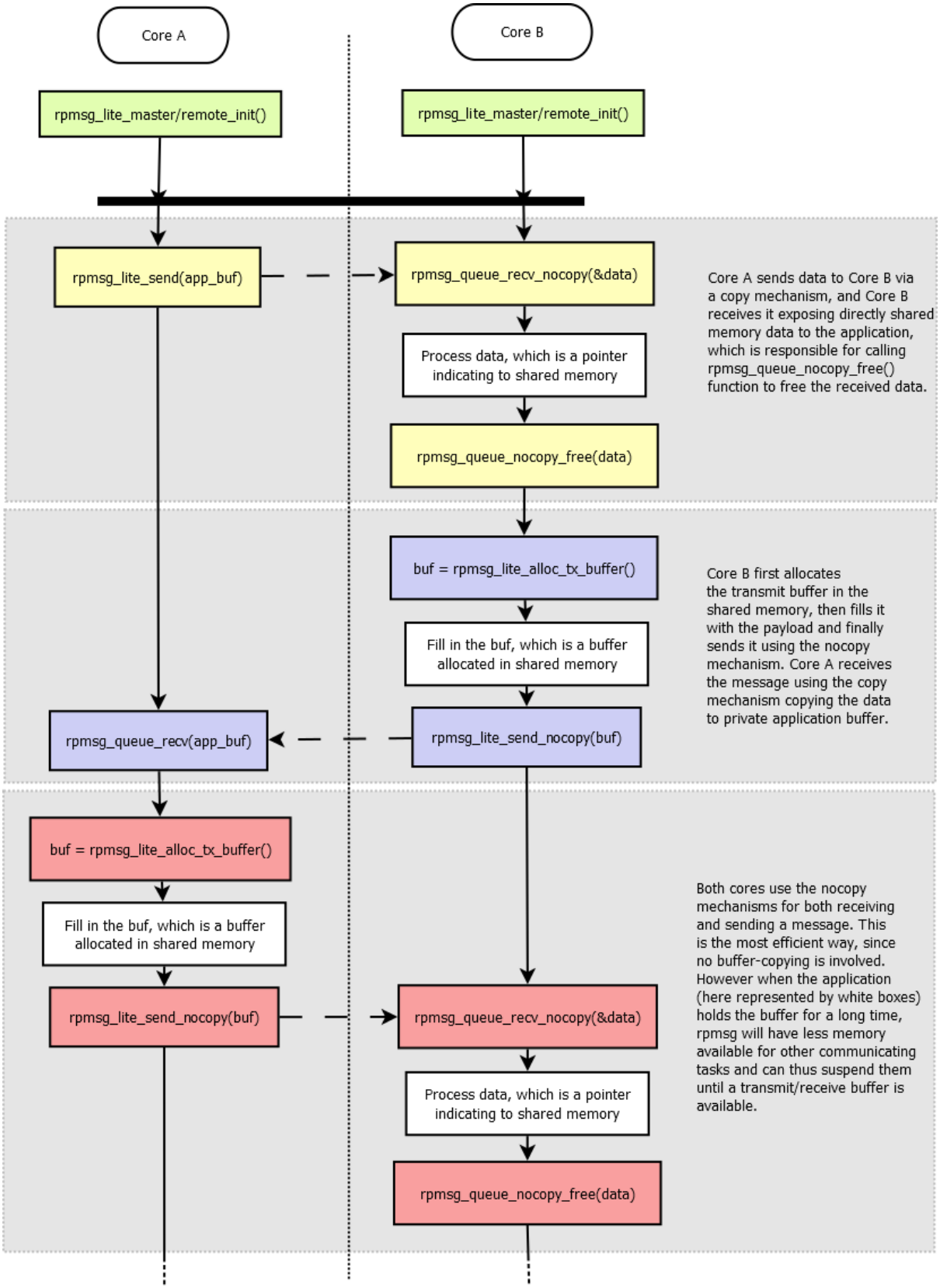
no-copy-send mechanism: This mechanism allows sending messages without the cost for copying data from the application buffer to the RPMsg/virtio buffer in the shared memory. The sequence of no-copy sending steps to be performed is as follows:

- Call the rpmsg_lite_alloc_tx_buffer() function to get the virtio buffer and provide the buffer pointer to the application.
- Fill the data to be sent into the pre-allocated virtio buffer. Ensure that the filled data does not exceed the buffer size (provided as the rpmsg_lite_alloc_tx_buffer() size output parameter).
- Call the rpmsg_lite_send_nocopy() function to send the message to the destination endpoint. Consider the cache functionality and the virtio buffer alignment. See the rpmsg_lite_send_nocopy() function description below.

no-copy-receive mechanism: This mechanism allows reading messages without the cost for copying data from the virtio buffer in the shared memory to the application buffer. The sequence of no-copy receiving steps to be performed is as follows:

- Call the rpmsg_queue_recv_nocopy() function to get the virtio buffer pointer to the received data.
- Read received data directly from the shared memory.
- Call the rpmsg_queue_nocopy_free() function to release the virtio buffer and to make it available for the next data transfer.

The user is responsible for destroying any RPMsg-Lite objects he has created in case of deinitialization. In order to do this, the function rpmsg_queue_destroy() is used to destroy a queue, rpmsg_lite_destroy_ept() is used to destroy an endpoint and finally, rpmsg_lite_deinit() is used to deinitialize the RPMsg-Lite intercore communication stack. Deinitialize all endpoints using a queue before deinitializing the queue. Otherwise, you are actively invalidating the used queue handle, which is not allowed. RPMsg-Lite does not check this internally, since its main aim is to be lightweight.



Examples RPMsg_Lite multicore examples are part of NXP MCUXpressoSDK packages. Visit https://mcuxpresso.nxp.com to configure, build and download these packages. To get the board list with multicore support (RPMsg_Lite included) use filtering based on Middleware and search for 'multicore' string. Once the selected package with the multicore middleware is downloaded,

see

<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples for RPMsg_Lite multicore examples with 'rpmsg_lite_' name prefix.

Another way of getting NXP MCUXpressoSDK RPMsg_Lite multicore examples is using the mcuxsdk-manifests Github repo. Follow the description how to use the West tool to clone and update the mcuxsdk-manifests repo in readme section. Once done the armgcc rpmsg_lite examples can be found in

mcuxsdk/examples/_<board_name>/multicore_examples

You can use the evkmimxrt1170 as the board_name for instance. Similar to MCUXpressoSDK packages the RPMsg_Lite examples use the 'rpmsg_lite_' name prefix.

Notes

Environment layers implementation Several environment layers are provided in lib/rpmsg_lite/porting/environment folder. Not all of them are fully tested however. Here is the list of environment layers that passed testing:

- rpmsg_env_bm.c
- rpmsg_env_freertos.c
- rpmsg_env_xos.c
- rpmsg_env_threadx.c

The rest of environment layers has been created and used in some experimental projects, it has been running well at the time of creation but due to the lack of unit testing there is no guarantee it is still fully functional.

Shared memory configuration It is important to correctly initialize/configure the shared memory for data exchange in the application. The shared memory must be accessible from both the master and the remote core and it needs to be configured as Non-Cacheable memory. Dedicated shared memory section in liker file is also a good practise, it is recommended to use linker files from MCUXpressSDK packages for NXP devices based applications. It needs to be ensured no other application part/component is unintentionally accessing this part of memory.

Configuration options The RPMsg-Lite can be configured at the compile time. The default configuration is defined in the rpmsg_default_config.h header file. This configuration can be customized by the user by including rpmsg_config.h file with custom settings. The following table summarizes all possible RPMsg-Lite configuration options.

Configuration option	Default value	Usage
RL_MS_PE	(1)	Delay in milliseconds used in non-blocking API functions for polling.
RL_BUFFE	(496)	Size of the buffer payload, it must be more than 1 byte, and has to be word align (including rpmsg header size 16 bytes), if not it will be aligned up
RL_BUFFE	(2)	Number of the buffers, it must be power of two (2, 4, ...)
RL_API_H	(1)	Zero-copy API functions enabled/disabled.
RL_USE_S	(0)	Static API functions (no dynamic allocation) enabled/disabled.
RL_USE_D	(0)	Memory cache management of shared memory. Use in case of data cache is enabled for shared memory.
RL_CLEAF	(0)	Clearing used buffers before returning back to the pool of free buffers enabled/disabled.
RL_USE_M	(0)	When enabled IPC interrupts are managed by the Multicore Manager (IPC interrupts router), when disabled RPMsg-Lite manages IPC interrupts by itself.
RL_USE_E	(0)	When enabled the environment layer uses its own context. Required for some environments (QNX). The default value is 0 (no context, saves some RAM).
RL_DEBU(	(0)	When enabled buffer pointers passed to rpmsg_lite_send_nocopy() and rpmsg_lite_release_rx_buffer() functions (enabled by RL_API_HAS_ZEROCOPY config) are checked to avoid passing invalid buffer pointer. The default value is 0 (disabled). Do not use in RPMsg-Lite to Linux configuration.
RL_ALLO\	(0)	When enabled the opposite side is notified each time received buffers are consumed and put into the queue of available buffers. Enable this option in RPMsg-Lite to Linux configuration to allow unblocking of the Linux blocking send. The default value is 0 (RPMsg-Lite to RPMsg-Lite communication).
RL_ALLO\	(0)	It allows to define custom shared memory configuration and replacing the shared memory related global settings from rpmsg_config.h This is useful when multiple instances are running in parallel but different shared memory arrangement (vring size & alignment, buffers size & count) is required. The default value is 0 (all RPMsg_Lite instances use the same shared memory arrangement as defined by common config macros).
RL_ASSER	see rpmsg	Assert implementation.

How to format rpmsg-lite code To format code, use the application developed by Google, named *clang-format*. This tool is part of the llvm project. Currently, the clang-format 10.0.0 version is used for rpmsg-lite. The set of style settings used for clang-format is defined in the `.clang-format` file, placed in a root of the rpmsg-lite directory where Python script `run_clang_format.py` can be executed. This script executes the application named *clang-format.exe*. You need to have the path of this application in the OS's environment path, or you need to change the script.

References

[1] M. Novak, M. Cingel, Lockless Shared Memory Based Multicore Communication Protocol
Copyright © 2016 Freescale Semiconductor, Inc. Copyright © 2016-2025 NXP

Changelog RPMSG-Lite All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

[v5.3.0]

Added

- RT700 porting layer added support to send rpmsg messages between CM33_0 <-> Hifi1 and CM33_1 <-> Hifi4 cores.
- Add new platform macro RL_PLATFORM_MAX_ISR_COUNT this will set number of IRQ count per platform. This macro is then used in environment layers to set isr_table size where irq handles are registered. It size should match the bit length of VQ_ID so all combinations can fit into table.
- Unit tests updated to improve code coverage, new unit tests added covering static allocations in rtos environment layers.

Fixed

- virtio.h removed typedef uint8_t boolean and in its place use standard C99 bool type to avoid potential type conflicts.
- env_acquire_sync_lock() and env_release_sync_lock() synchronization primitives removed
- Kconfig consolidation, when RL_ALLOW_CUSTOM_SHMEM_CONFIG enabled the platform_get_custom_shmem_config() function needs to be implemented in platform layer to provide custom shared memory configuration for RPMsg-Lite instance.

v5.2.1

Added

- Doc added RPMSG-Lite VirtIO Overview
- Doc added RPSMG-Lite Design Considerations
- Added frdmimxrt1186 unit testing

Changed

- Remove limitation that RL_BUFFER_SIZE needs to be power of 2. It just has to be more than 16 bytes, e.g. 16 bytes of rpmsg header and payload size at least 1 byte and word aligned, if not it will be aligned up.

Fixed

- Fixed CERT-C INT31-C violation in platform_notify function in rpmsg_platform.c for imxrt700_m33, imxrt700_hifi4, imxrt700_hifi1 platforms

v5.2.0

Added

- Add MCXL20 porting layer and unit testing
- New utility macro RL_CALCULATE_BUFFER_COUNT_DOWN_SAFE to safely determine maximum buffer count within shared memory while preventing integer underflow.
- RT700 platform add support for MCMGR in DSPs

Changed

- Change rpmsg_platform.c to support new MCMGR API
- Improved input validation in initialization functions to properly handle insufficient memory size conditions.
- Refactored repeated buffer count calculation pattern for better code maintainability.
- To make sure that remote has already registered IRQ there is required App level IPC mechanism to notify master about it

Fixed

- Fixed env_wait_for_link_up function to handle timeout in link state checks for baremetal and qnx environment, RL_BLOCK mode can be used to wait indefinitely.
- Fixed CERT-C INT31-C violation by adding compile-time check to ensure RL_PLATFORM_HIGHEST_LINK_ID remains within safe range for 16-bit casting in virtqueue ID creation.
- Fixed CERT-C INT30-C violations by adding protection against unsigned integer underflow in shared memory calculations, specifically in shmem_length - (uint32_t)RL_VRING_OVERHEAD and shmem_length - 2U * shmem_config.vring_size expressions.
- Fixed CERT INT31-C violation in platform_interrupt_disable() and similar functions by replacing unsafe cast from uint32_t to int32_t with a return of 0 constant.
- Fixed unsigned integer underflow in rpmsg_lite_alloc_tx_buffer() where subtracting header size from buffer size could wrap around if buffer was too small, potentially leading to incorrect buffer sizing.
- Fixed CERT-C INT31-C violation in rpmsg_lite.c where size parameter was cast from uint32_t to uint16_t without proper validation.
  - Applied consistent masking approach to both size and flags parameters: (uint16_t)(value & 0xFFFFU).
  - This fix prevents potential data loss when size values exceed 65535.
- Fixed CERT INT31-C violation in env_memset functions by explicitly converting int32_t values to unsigned char using bit masking. This prevents potential data loss or misinterpretation when passing values outside the unsigned char range (0-255) to the standard memset() function.
- Fixed CERT-C INT31-C violations in RPMsg-Lite environment porting: Added validation checks for signed-to-unsigned integer conversions to prevent data loss and misinterpretation.
  - rpmsg_env_freertos.c: Added validation before converting int32_t to UBaseType_t.
  - rpmsg_env_qnx.c: Fixed format string and added validation before assigning to mqstat fields.
  - rpmsg_env_threadx.c: Added validation to prevent integer overflow and negative values.
  - rpmsg_env_xos.c: Added range checking before casting to uint16_t.
  - rpmsg_env_zephyr.c: Added validation before passing values to k_msgq_init.
- Fixed a CERT INT31-C compliance issue in env_get_current_queue_size() function where an unsigned queue count was cast to a signed int32_t without proper validation, which could lead to lost or misinterpreted data if queue size exceeded INT32_MAX.
- Fixed CERT INT31-C violation in rpmsg_platform.c where memcmp() return value (signed int) was compared with unsigned constant without proper type handling.

- Fixed CERT INT31-C violation in rpmsg_platform.c where casting from uint32_t to uint16_t could potentially result in data loss. Changed length variable type from uint16_t to uint32_t to properly handle memory address differences without truncation.
- Fixed potential integer overflow in env_sleep_msec() function in ThreadX environment implementation by rearranging calculation order in the sleep duration formula.
- Fixed CERT-C INT31-C violation in RPMsg-Lite where bitwise NOT operations on integer constants were performed in signed integer context before being cast to unsigned. This could potentially lead to misinterpreted data on imx943 platform.
- Added RL_MAX_BUFFER_COUNT (32768U) and RL_MAX_VRING_ALIGN (65536U) limit to ensure alignment values cannot contribute to integer overflow
- Fixed CERT INT31-C violation in vring_need_event(), added cast to uint16_t for each operand.

v5.1.4 - 27-Mar-2025

Added

- Add KW43B43 porting layer

Changed

- Doxygen bump to version 1.9.6

v5.1.3 - 13-Jan-2025

Added

- Memory cache management of shared memory. Enable with #define RL_USE_DCACHE (1) in rpmsg_config.h in case of data cache is used.
- Cmake/Kconfig support added.
- Porting layers for imx95, imxrt700, mcmxw71x, mcmxw72x, kw47b42 added.

v5.1.2 - 08-Jul-2024

Changed

- Zephyr-related changes.
- Minor Misra corrections.

v5.1.1 - 19-Jan-2024

Added

- Test suite provided.
- Zephyr support added.

Changed

- Minor changes in platform and env. layers, minor test code updates.

v5.1.0 - 02-Aug-2023

Added

- RPMsg-Lite: Added aarch64 support.

Changed

- RPMsg-Lite: Increased the queue size to (2 * RL_BUFFER_COUNT) to cover zero copy cases.
- Code formatting using LLVM16.

Fixed

- Resolved issues in ThreadX env. layer implementation.

v5.0.0 - 19-Jan-2023

Added

- Timeout parameter added to rpmsg_lite_wait_for_link_up API function.

Changed

- Improved debug check buffers implementation - instead of checking the pointer fits into shared memory check the presence in the VirtIO ring descriptors list.
- VRING_SIZE is set based on number of used buffers now (as calculated in vring_init) - updated for all platforms that are not communicating to Linux rpmsg counterpart.

Fixed

- Fixed wrong RL_VRING_OVERHEAD macro comment in platform.h files
- Misra corrections.

v4.0.0 - 20-Jun-2022

Added

- Added support for custom shared memory arrangement per the RPMsg_Lite instance.
- Introduced new rpmsg_lite_wait_for_link_up() API function - this allows to avoid using busy loops in rtos environments, GitHub PR #21.

Changed

- Adjusted rpmsg_lite_is_link_up() to return RL_TRUE/RL_FALSE.

v3.2.0 - 17-Jan-2022

Added

- Added support for i.MX8 MP multicore platform.

Changed

- Improved static allocations - allow OS-specific objects being allocated statically, GitHub PR #14.
- Aligned rpmsg_env_xos.c and some platform layers to latest static allocation support.

Fixed

- Minor Misra and typo corrections, GitHub PR #19, #20.

v3.1.2 - 16-Jul-2021

Added

- Addressed MISRA 21.6 rule violation in rpmsg_env.h (use SDK's PRINTF in MCUXpressoSDK examples, otherwise stdio printf is used).
- Added environment layers for XOS.
- Added support for i.MX RT500, i.MX RT1160 and i.MX RT1170 multicore platforms.

Fixed

- Fixed incorrect description of the rpmsg_lite_get_endpoint_from_addr function.

Changed

- Updated RL_BUFFER_COUNT documentation (issue #10).
- Updated imxrt600_hifi4 platform layer.

v3.1.1 - 15-Jan-2021

Added

- Introduced RL_ALLOW_CONSUMED_BUFFERS_NOTIFICATION config option to allow opposite side notification sending each time received buffers are consumed and put into the queue of available buffers.
- Added environment layers for Threadx.
- Added support for i.MX8QM multicore platform.

Changed

- Several MISRA C-2012 violations addressed.

v3.1.0 - 22-Jul-2020

Added

- Added support for several new multicore platforms.

Fixed

- MISRA C-2012 violations fixed (7.4).
- Fixed missing lock in rpmsg_lite_rx_callback() for QNX env.
- Correction of rpmsg_lite_instance structure members description.
- Address -Waddress-of-packed-member warnings in GCC9.

Changed

- Clang update to v10.0.0, code re-formatted.

v3.0.0 - 20-Dec-2019

Added

- Added support for several new multicore platforms.

Fixed

- MISRA C-2012 violations fixed, incl. data types consolidation.
- Code formatted.

v2.2.0 - 20-Mar-2019

Added

- Added configuration macro RL_DEBUG_CHECK_BUFFERS.
- Several MISRA violations fixed.
- Added environment layers for QNX and Zephyr.
- Allow environment context required for some environment (controlled by the RL_USE_ENVIRONMENT_CONTEXT configuration macro).
- Data types consolidation.

v1.1.0 - 28-Apr-2017

Added

- Supporting i.MX6SX and i.MX7D MPU platforms.
- Supporting LPC5411x MCU platform.
- Baremental and FreeRTOS support.
- Support of copy and zero-copy transfer.
- Support of static API (without dynamic allocations).

Multicore Manager

MCUXpresso SDK : mcuxsdk-middleware-mcmgr (Multicore Manager)

Overview This repository is for MCUXpresso SDK Multicore Manager middleware delivery and it contains Multicore Manager component officially provided in NXP MCUXpresso SDK. This repository is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk for the complete delivery of MCUXpresso SDK to be able to build and run Multicore Manager examples that are based on mcux-sdk-middleware-mcmgr component.

Documentation Overall details can be reviewed here: MCUXpresso SDK Online Documentation

Visit Multicore Manager - Documentation to review details on the contents in this sub-repo.

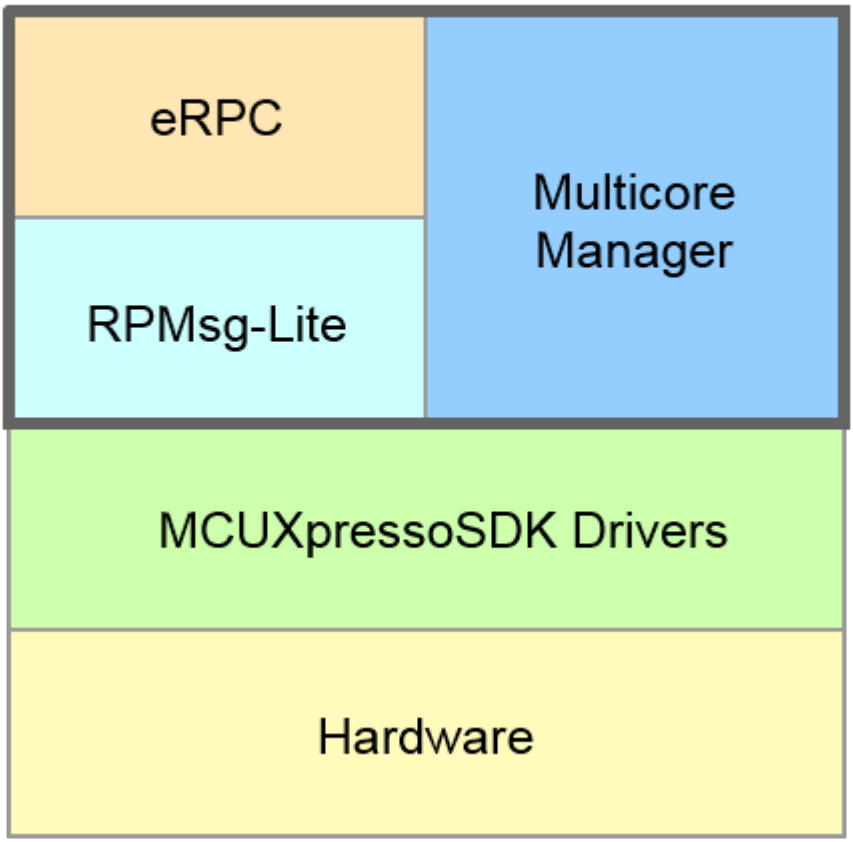
For Further API documentation, please look at doxygen documentation

Setup Instructions on how to install the MCUXpresso SDK provided from GitHub via west manifest Getting Started with SDK - Detailed Installation Instructions

Contribution We welcome and encourage the community to submit patches directly to the mcmgr project placed on github. Contributing can be managed via pull-requests. Before a pull-request is created the code should be tested and properly formatted.

Multicore Manager (MCMGR) The Multicore Manager (MCMGR) software library provides a number of services for multicore systems. This library is distributed as a part of the Multicore SDK (MCSDK). Together, the MCSDK and the MCUXpresso SDK (SDK) form a framework for development of software for NXP multicore devices.

The MCMGR component is located in the <MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr directory.



The Multicore Manager provides the following major functions:

- Maintains information about all cores in system.
- Secondary/auxiliary core(s) startup and shutdown.
- Remote core monitoring and event handling.

Usage of the MCMGR software component The main use case of MCMGR is the secondary/auxiliary core start. This functionality is performed by the public API function.

Example of MCMGR usage to start secondary core:

```
#include "mcmgr.h"

void main()
{
    /* Initialize MCMGR - low level multicore management library.
       Call this function as close to the reset entry as possible,
       (into the startup sequence) to allow CoreUp event triggering. */
    MCMGR_EarlyInit();

    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();
```

(continues on next page)

(continued from previous page)

```
    /* Boot secondary core application from the CORE1_BOOT_ADDRESS, pass "1" as startup data,␣
↪starting synchronously. */
    MCMGR_StartCore(kMCMGR_Core1, CORE1_BOOT_ADDRESS, 1, kMCMGR_Start_Synchronous);
.
.
.
    /* Stop secondary core execution. */
    MCMGR_StopCore(kMCMGR_Core1);
}
```

Some platforms allow stopping and re-starting the secondary core application again, using the MCMGR_StopCore / MCMGR_StartCore API calls. It is necessary to ensure the initially loaded image is not corrupted before re-starting, especially if it deals with the RAM target. Cache coherence has to be considered/ensured as well.

It could also happen that the secondary core application stops running correctly and the primary core application does not know about that situation. Therefore, it is beneficial to implement a mechanism for core health monitoring. The *test_heartbeat* unit test can serve as an example how to ensure that: secondary core could periodically send heartbeat signals to the primary core using MCMGR_TriggerEvent() API to indicate that it is alive and functioning properly.

Another important MCMGR feature is the ability for remote core monitoring and handling of events such as reset, exception, and application events. Application-specific callback functions for events are registered by the MCMGR_RegisterEvent() API. Triggering these events is done using the MCMGR_TriggerEvent() API. mcmgr_event_type_t enums all possible event types.

An example of MCMGR usage for remote core monitoring and event handling. Code for the primary side:

```
#include "mcmgr.h"

#define APP_RPMSG_READY_EVENT_DATA  (1)
#define APP_NUMBER_OF_CORES (2)
#define APP_SECONDARY_CORE kMCMGR_Core1

/* Callback function registered via the MCMGR_RegisterEvent() and triggered by MCMGR_TriggerEvent()␣
↪called on the secondary core side */
void RPMsgRemoteReadyEventHandler(mcmgr_core_t coreNum, uint16_t eventData, void *context)
{
    uint16_t *data = &((uint16_t *)context)[coreNum];

    *data = eventData;
}

void main()
{
    uint16_t RPMsgRemoteReadyEventData[NUMBER_OF_CORES] = {0};

    /* Initialize MCMGR - low level multicore management library.
       Call this function as close to the reset entry as possible,
       (into the startup sequence) to allow CoreUp event triggering. */
    MCMGR_EarlyInit();

    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();

    /* Register the application event before starting the secondary core */
    MCMGR_RegisterEvent(kMCMGR_RemoteApplicationEvent, RPMsgRemoteReadyEventHandler, (void␣
↪*)RPMsgRemoteReadyEventData);
```

(continues on next page)

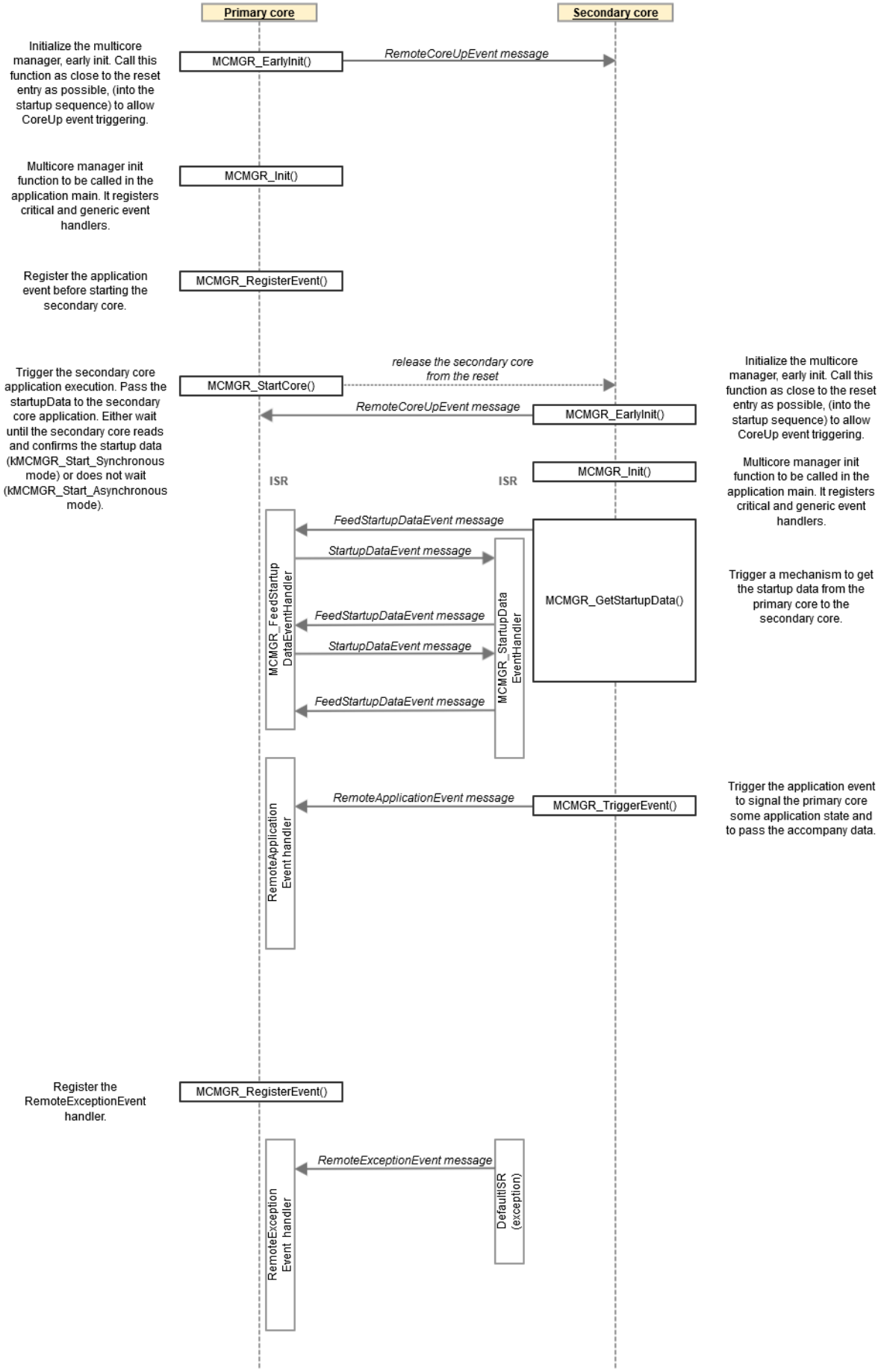
(continued from previous page)

```
    /* Boot secondary core application from the CORE1_BOOT_ADDRESS, pass rpmsg_lite_base address␣
↪as startup data, starting synchronously. */
    MCMGR_StartCore(APP_SECONDARY_CORE, CORE1_BOOT_ADDRESS, (uint32_t)rpmsg_lite_
↪base, kMCMGR_Start_Synchronous);

    /* Wait until the secondary core application signals the rpmsg remote has been initialized and is ready to␣
↪communicate. */
    while(APP_RPMSG_READY_EVENT_DATA != RPMsgRemoteReadyEventData[APP_SECONDARY_
↪CORE]) {};
.
.
.
}
```

Code for the secondary side:

```
#include "mcmgr.h"

#define APP_RPMSG_READY_EVENT_DATA  (1)

void main()
{
    /* Initialize MCMGR - low level multicore management library.
       Call this function as close to the reset entry as possible,
       (into the startup sequence) to allow CoreUp event triggering. */
    MCMGR_EarlyInit();

    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();
.
.
.

    /* Signal the to other core that we are ready by triggering the event and passing the APP_RPMSG_
↪READY_EVENT_DATA */
    MCMGR_TriggerEvent(kMCMGR_Core0, kMCMGR_RemoteApplicationEvent, APP_RPMSG_
↪READY_EVENT_DATA);
.
.
.
}
```

MCMGR Data Exchange Diagram The following picture shows how the handshakes are supposed to work between the two cores in the MCMGR software.


Primary core
Secondary core
Initialize the multicore manager, early init. Call this function as close to the reset entry as possible, (into the startup sequence) to allow CoreUp event triggering.
MCMGR_EarlyInit()
RemoteCoreUpEvent message
Multicore manager init function to be called in the application main. It registers critical and generic event handlers.
MCMGR_Init()
Register the application event before starting the secondary core.
MCMGR_RegisterEvent()
Trigger the secondary core application execution. Pass the startupData to the secondary core application. Either wait until the secondary core reads and confirms the startup data (kMCMGR_Start_Synchronous mode) or does not wait (kMCMGR_Start_Asynchronous mode).
MCMGR_StartCore()
release the secondary core from the reset
RemoteCoreUpEvent message
MCMGR_EarlyInit()
Initialize the multicore manager, early init. Call this function as close to the reset entry as possible, (into the startup sequence) to allow CoreUp event triggering.
MCMGR_Init()
Multicore manager init function to be called in the application main. It registers critical and generic event handlers.
ISR
ISR
FeedStartupDataEvent message
StartupDataEvent message
MCMGR_FeedStartup DataEventHandler
MCMGR_StartupData EventHandler
MCMGR_GetStartupData()
Trigger a mechanism to get the startup data from the primary core to the secondary core.
FeedStartupDataEvent message
StartupDataEvent message
FeedStartupDataEvent message
RemoteApplication Event handler
RemoteApplicationEvent message
MCMGR_TriggerEvent()
Trigger the application event to signal the primary core some application state and to pass the accompany data.
Register the RemoteExceptionEvent handler.
MCMGR_RegisterEvent()
RemoteException Event handler
RemoteExceptionEvent message
DefaultISR (exception)

Changelog Multicore Manager All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

[v5.0.2]

Added

- Added gcov options and configs to support mcmgr code coverage
- Added new test_weak_mu_isr testcase for devices with MU peripheral
- Added new test_heartbeat testcase showing heartbeat mechanism between primary and secondary cores using the MCMGR

v5.0.1

Added

- Added frdmimxrt1186 unit testing

Changed

- [KW43] Rename core#1 reset control register

Fixed

- Added CX flag into CMakeLists.txt to allow c++ build compatibility.
- Fix path to mcmgr headers directory in doxyfile

v5.0.0

Added

- Added MCMGR_BUSY_POLL_COUNT macro to prevent infinite polling loops in MCMGR operations.
- Implemented timeout mechanism for all polling loops in MCMGR code.
- Added support to handle more then two cores. Breaking API change by adding parameter coreNum specifying core number in functions bellow.
  - MCMGR_GetStartupData(uint32_t *startupData, mcmgr_core_t coreNum)
  - MCMGR_TriggerEvent(mcmgr_event_type_t type, uint16_t eventData, mcmgr_core_t coreNum)
  - MCMGR_TriggerEventForce(mcmgr_event_type_t type, uint16_t eventData, mcmgr_core_t coreNum)
  - typedef void (*mcmgr_event_callback_t)(uint16_t data, void *context, mcmgr_core_t coreNum);

When registering the event with function MCMGR_RegisterEvent() user now needs to provide callbackData pointer to array of elements per every core in system (see README.md for example).In case of systems with only two cores the coreNum in callback can be ignored as events can arrive only from one core. Please see Porting guide for more details: PortingGuideTo_v5.md

- Updated all porting files to support new MCMGR API.
- Added new platform specific include file mcmgr_platform.h. It will contain common platform specific macros that can be then used in mcmgr and application. e.g. platform core count MCMGR_CORECOUNT 4.
- Move all header files to new inc directory.
- Added new platform-specific include files inc/platform/<platform_name>/mcmgr_platform.h.

Added

- Add MCXL20 porting layer and unit testing

v4.1.7

Fixed

- mcmgr_stop_core_internal() function now returns kStatus_MCMGR_NotImplemented status code instead of kStatus_MCMGR_Success when device does not support stop of secondary core. Ports affected: kw32w1, kw45b41, kw45b42, mcxw716, mcxw727.

[v4.1.6]

Added

- Multicore Manager moved to standalone repository.
- Add porting layers for imxrt700, mcmxw727, kw47b42.
- New MCMGR_ProcessDeferredRxIsr() API added.

[v4.1.5]

Added

- Add notification into MCMGR_EarlyInit and mcmgr_early_init_internal functions to avoid using uninitialized data in their implementations.

[v4.1.4]

Fixed

- Avoid calling tx isr callbacks when respective Messaging Unit Transmit Interrupt Enable flag is not set in the CR/TCR register.
- Messaging Unit RX and status registers are cleared after the initialization.

[v4.1.3]

Added

- Add porting layers for imxrt1180.

Fixed

- mu_isr() updated to avoid calling tx isr callbacks when respective Transmit Interrupt Enable flag is not set in the CR/TCR register.
- mcmgr_mu_internal.c code adaptation to new supported SoCs.

[v4.1.2]

Fixed

- Update mcmgr_stop_core_internal() implementations to set core state to kMCMGR_ResetCoreState.

[v4.1.0]

Fixed

- Code adjustments to address MISRA C-2012 Rules

[v4.0.3]

Fixed

- Documentation updated to describe handshaking in a graphic form.
- Minor code adjustments based on static analysis tool findings

[v4.0.2]

Fixed

- Align porting layers to the updated MCUXpressoSDK feature files.

[v4.0.1]

Fixed

- Code formatting, removed unused code

[v4.0.0]

Added

- Add new MCMGR_TriggerEventForce() API.

[v3.0.0]

Removed

- Removed MCMGR_LoadApp(), MCMGR_MapAddress() and MCMGR_SignalReady()

Modified

- Modified MCMGR_GetStartupData()

Added

- Added MCMGR_EarlyInit(), MCMGR_RegisterEvent() and MCMGR_TriggerEvent()
- Added the ability for remote core monitoring and event handling

[v2.0.1]

Fixed

- Updated to be Misra compliant.

[v2.0.0]

Added

- Support for lpcxpresso54114 board.

[v1.1.0]

Fixed

- Ported to KSDK 2.0.0.

[v1.0.0]

Added

- Initial release.

eRPC

MCUXpresso SDK : mcuxsdk-middleware-erpc

Overview This repository is for MCUXpresso SDK eRPC middleware delivery and it contains eRPC component officially provided in NXP MCUXpresso SDK. This repository is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk for the complete delivery of MCUXpresso SDK to be able to build and run eRPC examples that are based on mcux-sdk-middleware-erpc component.

Documentation Overall details can be reviewed here: MCUXpresso SDK Online Documentation

Visit eRPC - Documentation to review details on the contents in this sub-repo.

Setup Instructions on how to install the MCUXpresso SDK provided from GitHub via west manifest Getting Started with SDK - Detailed Installation Instructions

Contribution We welcome and encourage the community to submit patches directly to the eRPC project placed on github. Contributing can be managed via pull-requests. Before a pull-request is created the code should be tested and properly formatted.

---

eRPC

- *MCUXpresso SDK : mcuxsdk-middleware-erpc*
  - *Overview*
  - *Documentation*
  - *Setup*
  - *Contribution*
- *eRPC*
  - *About*
  - *Releases*
    - *Edge releases*
  - *Documentation*
  - *Examples*
  - *References*
  - *Directories*
  - *Building and installing*
    - *Requirements*
      - *Windows*
      - *Mac OS X*
    - *Building*
      - *CMake and KConfig*
      - *Make*

* *Installing for Python*
- *Known issues and limitations*
- *Code providing*

About

eRPC (Embedded RPC) is an open source Remote Procedure Call (RPC) system for multichip embedded systems and heterogeneous multicore SoCs.

Unlike other modern RPC systems, such as the excellent Apache Thrift, eRPC distinguishes itself by being designed for tightly coupled systems, using plain C for remote functions, and having a small code size (<5kB). It is not intended for high performance distributed systems over a network.

eRPC does not force upon you any particular API style. It allows you to export existing C functions, without having to change their prototypes. (There are limits, of course.) And although the internal infrastructure is written in C++, most users will be able to use only the simple C setup APIs shown in the examples below.

A code generator tool called erpcgen is included. It accepts input IDL files, having an .erpc extension, that have definitions of your data types and remote interfaces, and generates the shim code that handles serialization and invocation. erpcgen can generate either C/C++ or Python code.

Example .erpc file:

```
// Define a data type.
enum LEDName { kRed, kGreen, kBlue }

// An interface is a logical grouping of functions.
interface IO {
    // Simple function declaration with an empty reply.
    set_led(LEDName whichLed, bool onOrOff) -> void
}
```

Client side usage:

```
void example_client(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_client_t client_manager;

    /* Init eRPC client infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    client_manager = erpc_client_init(transport, message_buffer_factory);

    /* init eRPC client IO service */
    initIO_client(client_manager);

    // Now we can call the remote function to turn on the green LED.
    set_led(kGreen, true);

    /* deinit objects */
    deinitIO_client();
    erpc_client_deinit(client_manager);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
```

(continues on next page)

(continued from previous page)

```
    erpc_transport_tcp_deinit(transport);
}
```

```
void example_client(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_client_t client_manager;

    /* Init eRPC client infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    client_manager = erpc_client_init(transport, message_buffer_factory);

    /* scope for client service */
    {
        /* init eRPC client IO service */
        IO_client client(client_manager);

        // Now we can call the remote function to turn on the green LED.
        client.set_led(kGreen, true);
    }

    /* deinit objects */
    erpc_client_deinit(client_manager);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
    erpc_transport_tcp_deinit(transport);
}
```

Server side usage:

```
// Implement the remote function.
void set_led(LEDName whichLed, bool onOrOff) {
    // implementation goes here
}

void example_server(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_server_t server;
    erpc_service_t service = create_IO_service();

    /* Init eRPC server infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    server = erpc_server_init(transport, message_buffer_factory);

    /* add custom service implementation to the server */
    erpc_add_service_to_server(server, service);

    // Run the server.
    erpc_server_run();

    /* deinit objects */
    destroy_IO_service(service);
    erpc_server_deinit(server);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
    erpc_transport_tcp_deinit(transport);
}
```

```
// Implement the remote function.
class IO : public IO_interface
```

(continues on next page)

(continued from previous page)

```
{
    /* eRPC call definition */
    void set_led(LEDName whichLed, bool onOrOff) override {
        // implementation goes here
    }
}

void example_server(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_server_t server;
    IO IOImpl;
    IO_service io(&IOImpl);

    /* Init eRPC server infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    server = erpc_server_init(transport, message_buffer_factory);

    /* add custom service implementation to the server */
    erpc_add_service_to_server(server, &io);

    /* poll for requests */
    erpc_status_t err = server.run();

    /* deinit objects */
    erpc_server_deinit(server);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
    erpc_transport_tcp_deinit(transport);
}
```

A number of transports are supported, and new transport classes are easy to write.

Supported transports can be found in *erpc/erpc_c/transport* folder. E.g:

- CMSIS UART
- NXP Kinetis SPI and DSPI
- POSIX and Windows serial port
- TCP/IP (mostly for testing)
- NXP RPMsg-Lite / RPMsg TTY
- SPIdev Linux
- USB CDC
- NXP Messaging Unit

eRPC is available with an unrestrictive BSD 3-clause license. See the LICENSE file for the full license text.

Releases eRPC releases

Edge releases Edge releases can by found on eRPC CircleCI webpage. Choose build of interest, then platform target and choose ARTIFACTS tab. Here you can find binary application from chosen build.

Documentation Documentation is in the wiki section.

eRPC Infrastructure documentation

Examples *Example IDL* is available in the *examples/* folder.

Plenty of eRPC multicore and multiprocessor examples can be also found in NXP MCUXpressoSDK packages. Visit https://mcuxpresso.nxp.com to configure, build and download these packages.

To get the board list with multicore support (eRPC included) use filtering based on Middleware and search for 'multicore' string. Once the selected package with the multicore middleware is downloaded, see

<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples for eRPC multicore examples (RPMsg_Lite or Messaging Unit transports used) or

<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples for eRPC multiprocessor examples (UART or SPI transports used).

eRPC examples use the 'erpc_' name prefix.

Another way of getting NXP MCUXpressoSDK eRPC multicore and multiprocessor examples is using the mcux-sdk Github repo. Follow the description how to use the West tool to clone and update the mcuxsdk repo in readme Overview section. Once done the armgcc eRPC examples can be found in

mcuxsdk/examples/<board_name>/multicore_examples or in

mcuxsdk/examples/<board_name>/multiprocessor_examples folders.

You can use the evkmimxrt1170 as the board_name for instance. Similar to MCUXpressoSDK packages the eRPC examples use the 'erpc_' name prefix.

References This section provides links to interesting erpc-based projects, articles, blogs or guides:

- erpc (EmbeddedRPC) getting started notes
- ERPC Linux Local Environment Construction and Use
- The New Wio Terminal eRPC Firmware

Directories *doc* - Documentation.

doxygen - Configuration and support files for running Doxygen over the eRPC C++ infrastructure and erpcgen code.

erpc_c - Holds C/C++ infrastructure for eRPC. This is the code you will include in your application.

erpc_python - Holds Python version of the eRPC infrastructure.

erpcgen - Holds source code for erpcgen and makefiles or project files to build erpcgen on Windows, Linux, and OS X.

erpcsniffer - Holds source code for erpcsniffer application.

examples - Several example IDL files.

mk - Contains common makefiles for building eRPC components.

test - Client/server tests. These tests verify the entire communications path from client to server and back.

utilities - Holds utilities which bring additional benefit to eRPC apps developers.

Building and installing These build instructions apply to host PCs and embedded Linux. For bare metal or RTOS embedded environments, you should copy the *erpc_c* directory into your application sources.

CMake and KConfig build:

It builds a static library of the eRPC C/C++ infrastructure, the erpcgen executable, and optionally the unit tests and examples.

CMake is compatible with gcc and clang. On Windows local MingGW downloaded by *script* can be used.

Make build:

It builds a static library of the eRPC C/C++ infrastructure, the erpcgen executable, and optionally the unit tests.

The makefiles are compatible with gcc or clang on Linux, OS X, and Cygwin. A Windows build of erpcgen using Visual Studio is also available in the *erpcgen/VisualStudio_v14* directory. There is also an Xcode project file in the *erpcgen* directory, which can be used to build erpcgen for OS X.

Requirements eRPC now support building **erpcgen**, **erpc_lib**, **tests** and **C examples** using CMake.

Requirements when using CMake:

- **CMake** (minimal version 3.20.0)
- Generator - **Make**, **Ninja**, ...
- **C/C++ compiler** - **GCC**, **CLANG**, ...
- **Binson** - https://www.gnu.org/software/bison/
- **Flex** - https://github.com/westes/flex/

Requirements when using Make:

- **Make**
- **C/C++ compiler** - **GCC**, **CLANG**, ...
- **Binson** - https://www.gnu.org/software/bison/
- **Flex** - https://github.com/westes/flex/

Windows Related steps to build **erpcgen** using **Visual Studio** are described in erpcgen/VisualStudio_v14/readme_erpcgen.txt.

To install MinGW, Bison, Flex locally on Windows:

```
./install_dependencies.ps1
* ```

#### Linux

```bash
./install_dependencies.sh
```

Mandatory for case, when build for different architecture is needed

- gcc-multilib, g++-multilib

Mac OS X
```

```
./install_dependencies.sh
```

Building

CMake and KConfig eRPC use CMake and KConfig to configurate and build eRPC related targets. KConfig can be edited by *prj.conf* or *menuconfig* when building.

Generate project, config and build. In *erpc/* execute:

```
cmake -B ./build # in erpc/build generate cmake project
cmake --build ./build --target menuconfig # Build menuconfig and configurate erpcgen, erpc_lib, tests and␣
↪examples
cmake --build ./build # Build all selected target from prj.conf/menuconfig
```

**CMake will use the system's default compilers and generator

If you want to use Windows and locally installed MinGW, use *CMake preset* :

```
cmake --preset mingw64 # Generate project in ./build using mingw64's make and compilers
cmake --build ./build --target menuconfig # Build menuconfig and configurate erpcgen, erpc_lib, tests and␣
↪examples
cmake --build ./build # Build all selected target from prj.conf/menuconfig
```

Make To build the library and erpcgen, run from the repo root directory:

```
make
```

To install the library, erpcgen, and include files, run:

```
make install
```

You may need to sudo the `make install`.

By default this will install into `/usr/local`. If you want to install elsewhere, set the `PREFIX` environment variable. Example for installing into `/opt`:

```
make install PREFIX=/opt
```

List of top level Makefile targets:

- `erpc`: build the liberpc.a static library
- `erpcgen`: build the erpcgen tool
- `erpcsniffer`: build the sniffer tool
- `test`: build the unit tests under the *test* directory
- `all`: build all of the above
- `install`: install liberpc.a, erpcgen, and include files

eRPC code is validated with respect to the C++ 11 standard.

Installing for Python To install the Python infrastructure for eRPC see instructions in the *erpc python readme*.

Known issues and limitations

- Static allocations controlled by the ERPC_ALLOCATION_POLICY config macro are not fully supported yet, i.e. not all erpc objects can be allocated statically now. It deals with the ongoing process and the full static allocations support will be added in the future.

Code providing Repository on Github contains two main branches: **main** and **develop**. Code is developed on **develop** branch. Release version is created via merging **develop** branch into **main** branch.

Copyright 2014-2016 Freescale Semiconductor, Inc.

Copyright 2016-2025 NXP

eRPC Getting Started

Overview This *Getting Started User Guide* shows software developers how to use Remote Procedure Calls (RPC) in embedded multicore microcontrollers (eRPC).

The eRPC documentation is located in the *<MCUXpressoSDK_install_dir>/ middleware/multicore/erpc/doc* folder.

Create an eRPC application This section describes a generic way to create a client/server eRPC application:

1. **Design the eRPC application:** Decide which data types are sent between applications, and define functions that send/receive this data.
2. **Create the IDL file:** The IDL file contains information about data types and functions used in an eRPC application, and is written in the IDL language.
3. **Use the eRPC generator tool:** This tool takes an IDL file and generates the shim code for the client and the server-side applications.
4. **Create an eRPC application:**
    1. Create two projects, where one project is for the client side (primary core) and the other project is for the server side (secondary core).
    2. Add generated files for the client application to the client project, and add generated files for the server application to the server project.
    3. Add infrastructure files.
    4. Add user code for client and server applications.
    5. Set the client and server project options.
5. **Run the eRPC application:** Run both the server and the client applications. Make sure that the server has been run before the client request was sent.

A specific example follows in the next section.

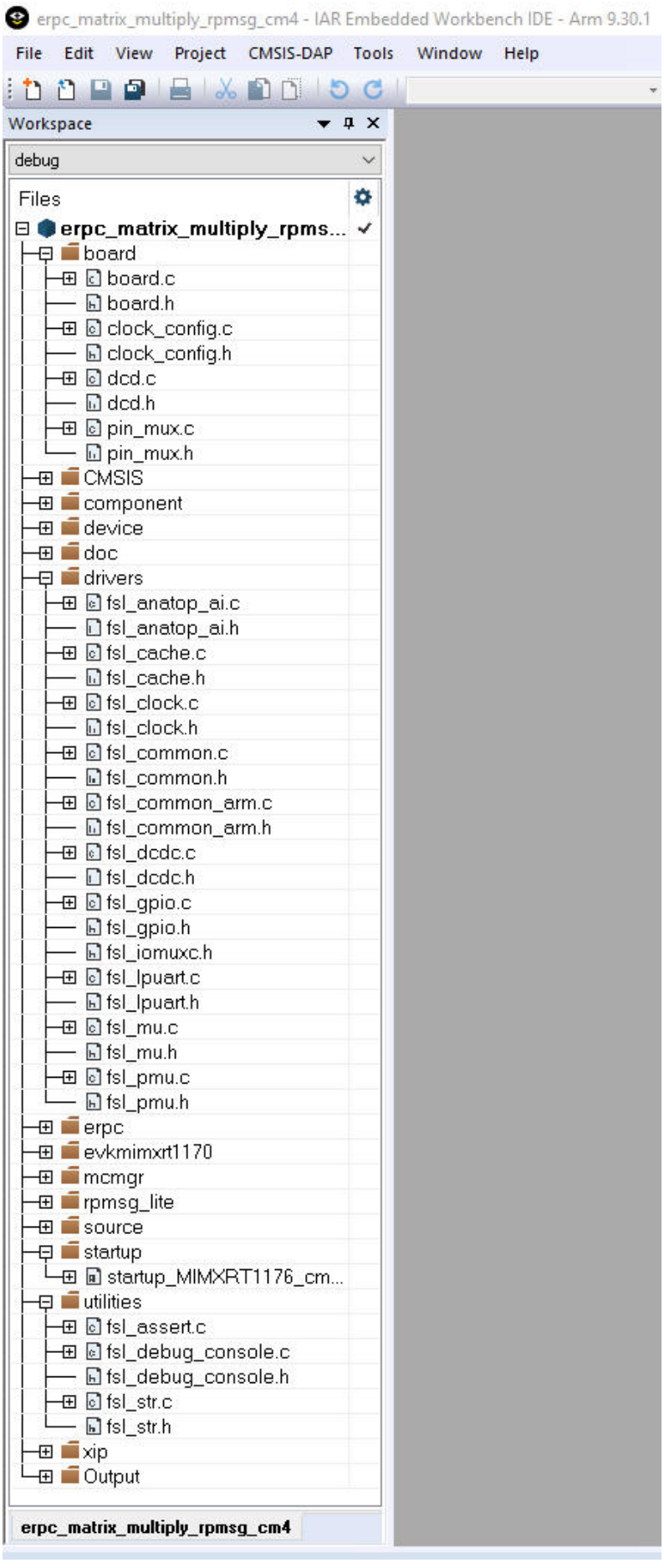
Multicore server application The "Matrix multiply" eRPC server project is located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm4/iar

The project files for the eRPC server have the `_cm4` suffix.

Server project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in:

- *<MCUXpressoSDK_install_dir>/devices/<device>*
- *<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/*


erpc_matrix_multiply_rpmsg_cm4 - IAR Embedded Workbench IDE - Arm 9.30.1
File Edit View Project CMSIS-DAP Tools Window Help
Workspace
debug
Files
erpc_matrix_multiply_rpms...
board
board.c
board.h
clock_config.c
clock_config.h
dcd.c
dcd.h
pin_mux.c
pin_mux.h
CMSIS
component
device
doc
drivers
fsl_anatop_ai.c
fsl_anatop_ai.h
fsl_cache.c
fsl_cache.h
fsl_clock.c
fsl_clock.h
fsl_common.c
fsl_common.h
fsl_common_arm.c
fsl_common_arm.h
fsl_dcdc.c
fsl_dcdc.h
fsl_gpio.c
fsl_gpio.h
fsl_iomuxc.h
fsl_lpuart.c
fsl_lpuart.h
fsl_mu.c
fsl_mu.h
fsl_pmu.c
fsl_pmu.h
erpc
evkmimxrt1170
mcmgr
rpmsg_lite
source
startup
startup_MIMXRT1176_cm...
utilities
fsl_assert.c
fsl_debug_console.c
fsl_debug_console.h
fsl_str.c
fsl_str.h
xip
Output
erpc_matrix_multiply_rpmsg_cm4

|

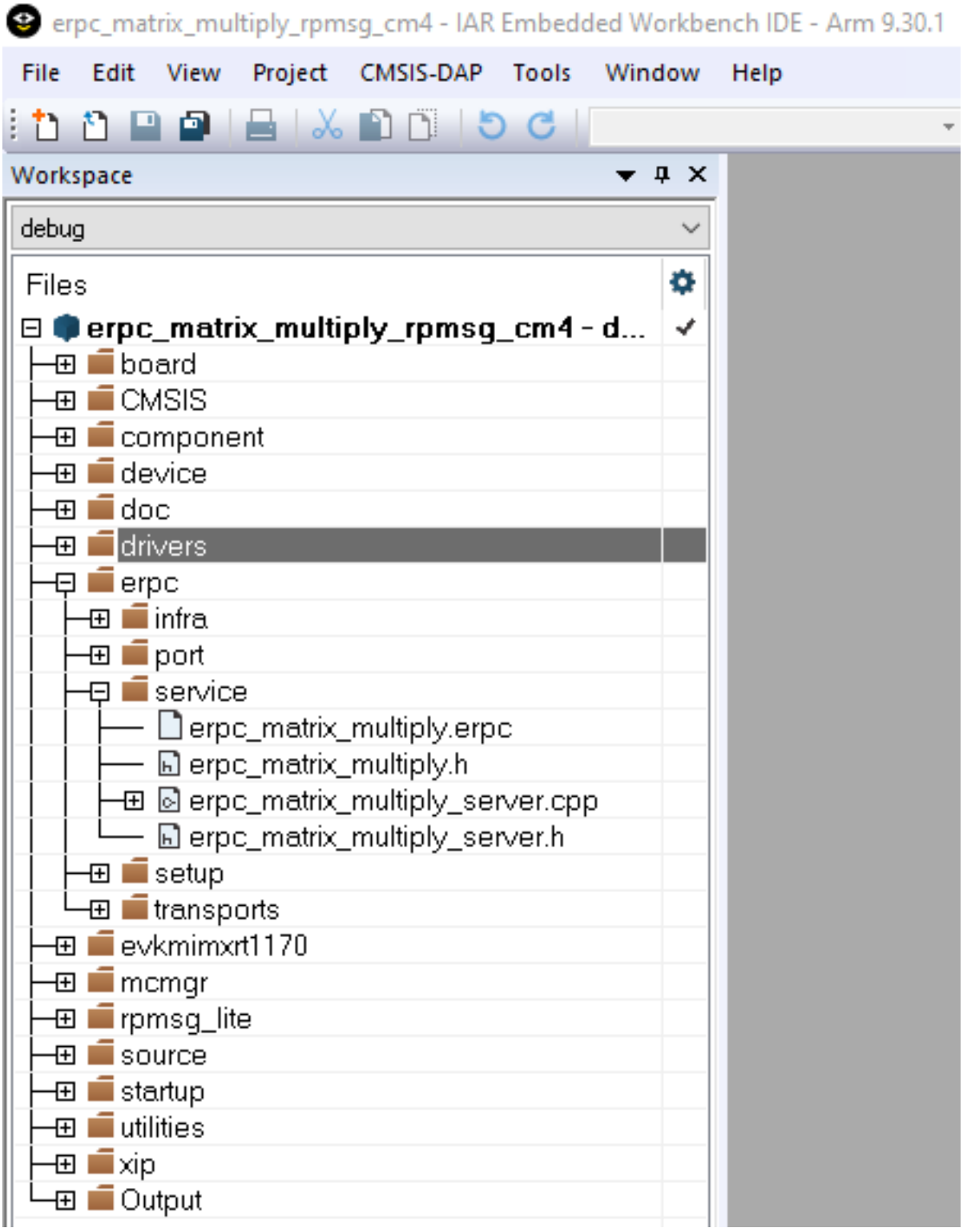
Parent topic: Multicore server application

Server related generated files The server-related generated files are:

- erpc_matric_multiply.h
- erpc_matrix_multiply_server.h
- erpc_matrix_multiply_server.cpp

The server-related generated files contain the shim code for functions and data types declared in the IDL file. These files also contain functions for the identification of client requested functions, data deserialization, calling requested function's implementations, and data serialization and return, if requested by the client. These shim code files can be found in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/s



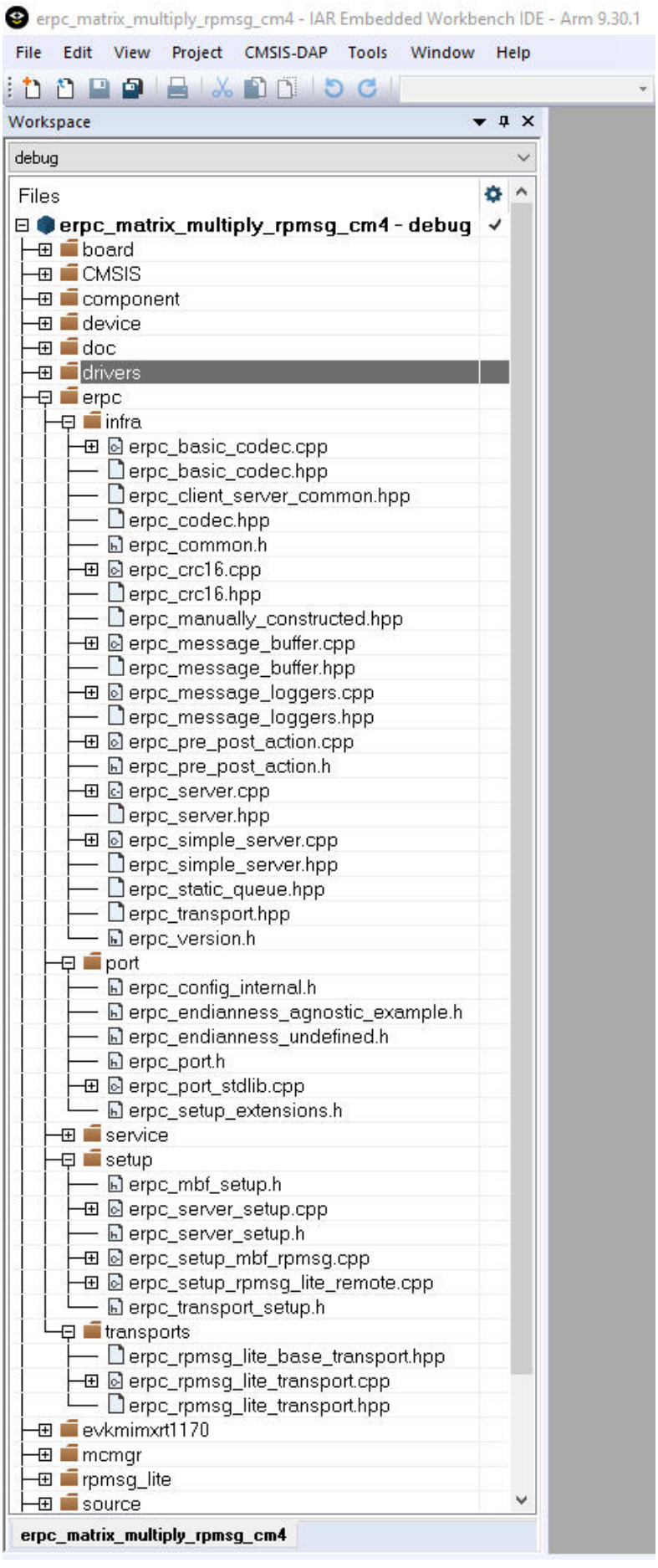
Parent topic: Multicore server application

Server infrastructure files The eRPC infrastructure files are located in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c

The **erpc_c** folder contains files for creating eRPC client and server applications in the *C/C++* language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.
  - Four files, erpc_server.hpp, erpc_server.cpp, erpc_simple_server.hpp, and erpc_simple_server.cpp, are used for running the eRPC server on the server-side applications. The simple server is currently the only implementation of the server, and its role is to catch client requests, identify and call requested functions, and send data back when requested.
  - Three files (erpc_codec.hpp, erpc_basic_codec.hpp, and erpc_basic_codec.cpp) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
  - The erpc_common.hpp file is used for common eRPC definitions, typedefs, and enums.
  - The erpc_manually_constructed.hpp file is used for allocating static storage for the used objects.
  - Message buffer files are used for storing serialized data: erpc_message_buffer.h and erpc_message_buffer.cpp.
  - The erpc_transport.h file defines the abstract interface for transport layer.
- The **port** subfolder contains the eRPC porting layer to adapt to different environments.
  - erpc_port.h file contains definition of erpc_malloc() and erpc_free() functions.
  - erpc_port_stdlib.cpp file ensures adaptation to stdlib.
  - erpc_config_internal.h internal erpc configuration file.
- The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.
  - The erpc_server_setup.h and erpc_server_setup.cpp files needs to be added into the "Matrix multiply" example project to demonstrate the use of C-wrapped functions in this example.
  - The erpc_transport_setup.h and erpc_setup_rpmsg_lite_remote.cpp files needs to be added into the project in order to allow the C-wrapped function for transport layer setup.
  - The erpc_mbf_setup.h and erpc_setup_mbf_rpmsg.cpp files needs to be added into the project in order to allow message buffer factory usage.
- The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions in the setup folder.
  - RPMsg-Lite is used as the transport layer for the communication between cores, erpc_rpmsg_lite_base_transport.hpp, erpc_rpmsg_lite_transport.hpp, and erpc_rpmsg_lite_transport.cpp files need to be added into the server project.


erpc_matrix_multiply_rpmsg_cm4 - IAR Embedded Workbench IDE - Arm 9.30.1
File Edit View Project CMSIS-DAP Tools Window Help
Workspace
debug
Files
erpc_matrix_multiply_rpmsg_cm4 - debug
board
CMSIS
component
device
doc
drivers
erpc
infra
erpc_basic_codec.cpp
erpc_basic_codec.hpp
erpc_client_server_common.hpp
erpc_codec.hpp
erpc_common.h
erpc_crc16.cpp
erpc_crc16.hpp
erpc_manually_constructed.hpp
erpc_message_buffer.cpp
erpc_message_buffer.hpp
erpc_message_loggers.cpp
erpc_message_loggers.hpp
erpc_pre_post_action.cpp
erpc_pre_post_action.h
erpc_server.cpp
erpc_server.hpp
erpc_simple_server.cpp
erpc_simple_server.hpp
erpc_static_queue.hpp
erpc_transport.hpp
erpc_version.h
port
erpc_config_internal.h
erpc_endianness_agnostic_example.h
erpc_endianness_undefined.h
erpc_port.h
erpc_port_stdlib.cpp
erpc_setup_extensions.h
service
setup
erpc_mbf_setup.h
erpc_server_setup.cpp
erpc_server_setup.h
erpc_setup_mbf_rpmsg.cpp
erpc_setup_rpmsg_lite_remote.cpp
erpc_transport_setup.h
transports
erpc_rpmsg_lite_base_transport.hpp
erpc_rpmsg_lite_transport.cpp
erpc_rpmsg_lite_transport.hpp
evkmimxrt1170
mcmgr
rpmsg_lite
source
erpc_matrix_multiply_rpmsg_cm4

|

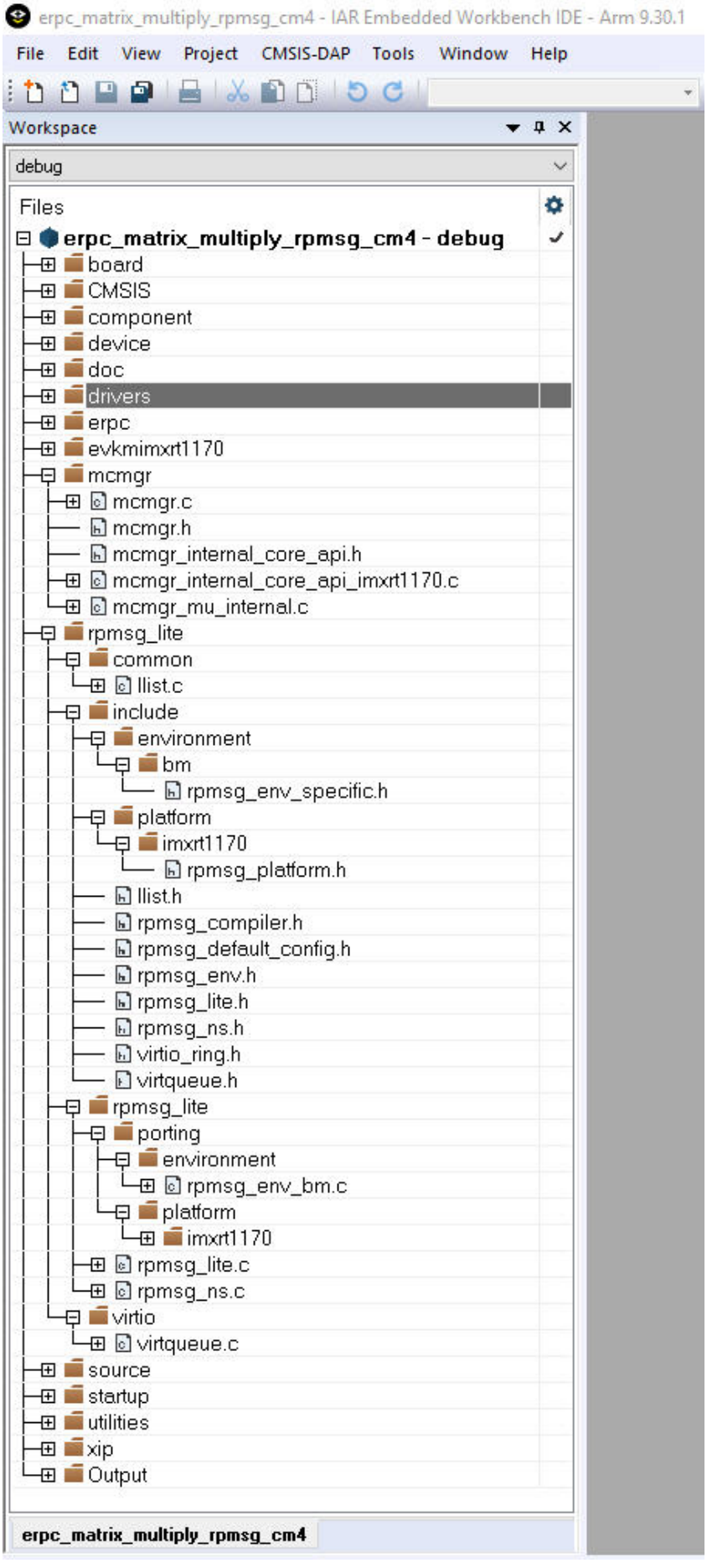
Parent topic: Multicore server application

Server multicore infrastructure files Because of the RPMsg-Lite (transport layer), it is also necessary to include RPMsg-Lite related files, which are in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/


erpc_matrix_multiply_rpmsg_cm4 - IAR Embedded Workbench IDE - Arm 9.30.1
File Edit View Project CMSIS-DAP Tools Window Help
Workspace
debug
Files
erpc_matrix_multiply_rpmsg_cm4 - debug
board
CMSIS
component
device
doc
drivers
erpc
evkmimxrt1170
mcmgr
mcmgr.c
mcmgr.h
mcmgr_internal_core_api.h
mcmgr_internal_core_api_imxrt1170.c
mcmgr_mu_internal.c
rpmsg_lite
common
llist.c
include
environment
bm
rpmsg_env_specific.h
platform
imxrt1170
rpmsg_platform.h
llist.h
rpmsg_compiler.h
rpmsg_default_config.h
rpmsg_env.h
rpmsg_lite.h
rpmsg_ns.h
virtio_ring.h
virtqueue.h
rpmsg_lite
porting
environment
rpmsg_env_bm.c
platform
imxrt1170
rpmsg_lite.c
rpmsg_ns.c
virtio
virtqueue.c
source
startup
utilities
xip
Output
erpc_matrix_multiply_rpmsg_cm4

|

Parent topic: Multicore server application

Server user code The server's user code is stored in the main_core1.c file, located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm4

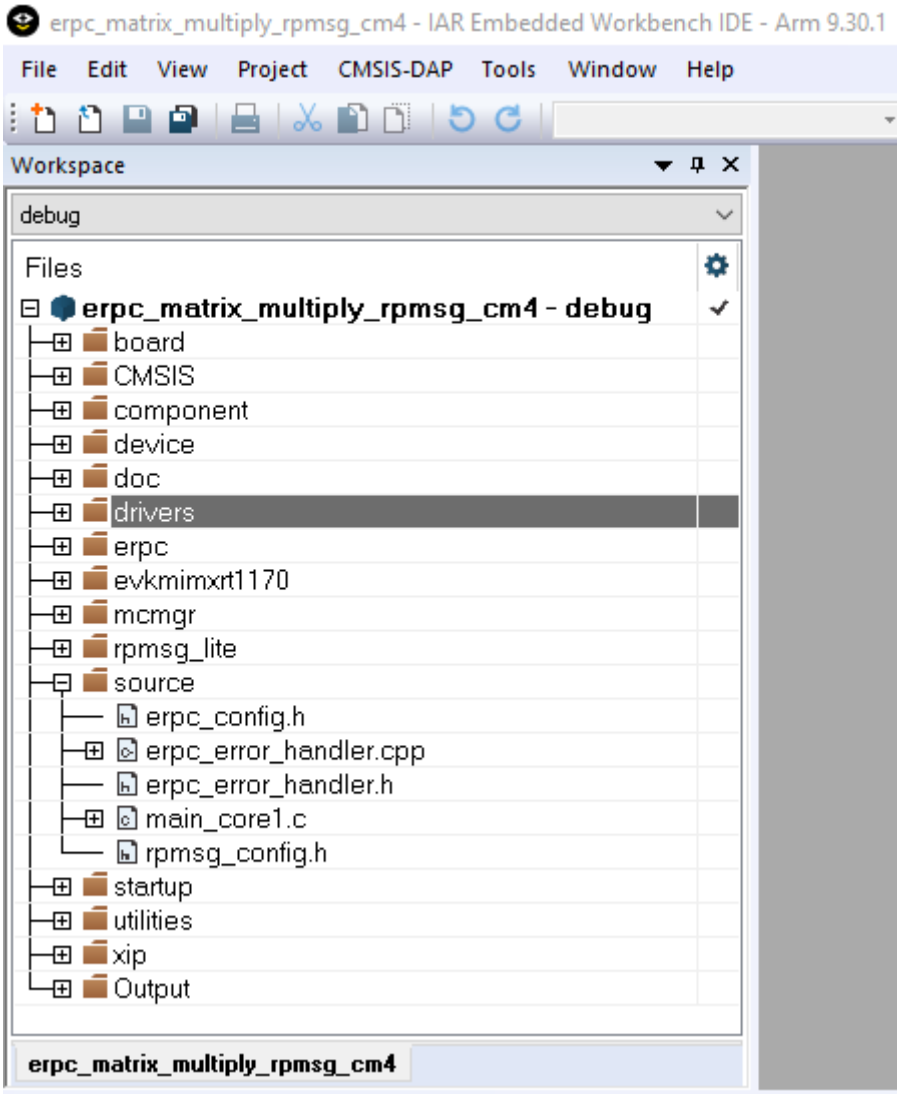
The main_core1.c file contains two functions:

- The **main()** function contains the code for the target board and eRPC server initialization. After the initialization, the matrix multiply service is added and the eRPC server waits for client's requests in the while loop.
- The **erpcMatrixMultiply()** function is the user implementation of the eRPC function defined in the IDL file.
- There is the possibility to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in the erpc_error_handler.h and erpc_error_handler.cpp files.

The eRPC-relevant code is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(const Matrix *matrix1, const Matrix *matrix2, Matrix *result_matrix)
{
  ...
}
int main()
{
  ...
  /* RPMsg-Lite transport layer initialization */
  erpc_transport_t transport;
  transport = erpc_transport_rpmsg_lite_remote_init(src, dst, (void*)startupData,
  ERPC_TRANSPORT_RPMSG_LITE_LINK_ID, SignalReady, NULL);
  ...
  /* MessageBufferFactory initialization */
  erpc_mbf_t message_buffer_factory;
  message_buffer_factory = erpc_mbf_rpmsg_init(transport);
  ...
  /* eRPC server side initialization */
  erpc_server_t server;
  server = erpc_server_init(transport, message_buffer_factory);
  ...
  /* Adding the service to the server */
  erpc_service_t service = create_MatrixMultiplyService_service();
  erpc_add_service_to_server(server, service);
  ...
  while (1)
  {
    /* Process eRPC requests */
    erpc_status_t status = erpc_server_poll(server);
    /* handle error status */
    if (status != kErpcStatus_Success)
    {
      /* print error description */
      erpc_error_handler(status, 0);
      ...
    }
    ...
  }
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the *<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/ erpc_matrix_multiply_rpmsg* folder.



Parent topic: Multicore server application

Parent topic:*Create an eRPC application*

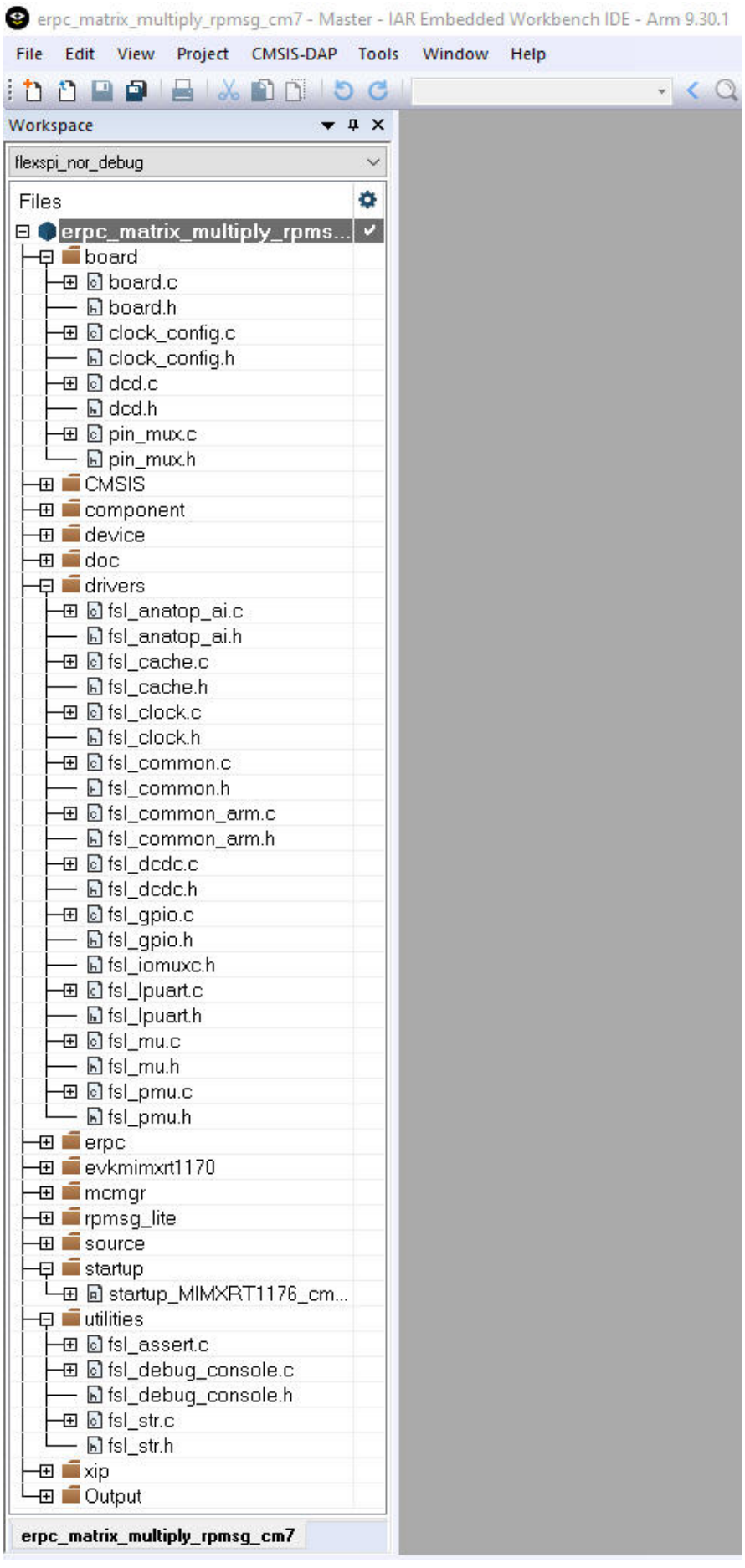
Multicore client application The "Matrix multiply" eRPC client project is located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm7/iar

Project files for the eRPC client have the _cm7 suffix.

Client project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in the following folders:

- *<MCUXpressoSDK_install_dir>/devices/<device>*
- *<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/*


erpc_matrix_multiply_rpmsg_cm7 - Master - IAR Embedded Workbench IDE - Arm 9.30.1
File
Edit
View
Project
CMSIS-DAP
Tools
Window
Help
Workspace
flexspi_nor_debug
Files
erpc_matrix_multiply_rpms...
board
board.c
board.h
clock_config.c
clock_config.h
dcd.c
dcd.h
pin_mux.c
pin_mux.h
CMSIS
component
device
doc
drivers
fsl_anatop_ai.c
fsl_anatop_ai.h
fsl_cache.c
fsl_cache.h
fsl_clock.c
fsl_clock.h
fsl_common.c
fsl_common.h
fsl_common_arm.c
fsl_common_arm.h
fsl_dcdc.c
fsl_dcdc.h
fsl_gpio.c
fsl_gpio.h
fsl_iomuxc.h
fsl_lpuart.c
fsl_lpuart.h
fsl_mu.c
fsl_mu.h
fsl_pmu.c
fsl_pmu.h
erpc
evkmimxrt1170
mcmgr
rpmsg_lite
source
startup
startup_MIMXRT1176_cm...
utilities
fsl_assert.c
fsl_debug_console.c
fsl_debug_console.h
fsl_str.c
fsl_str.h
xip
Output
erpc_matrix_multiply_rpmsg_cm7

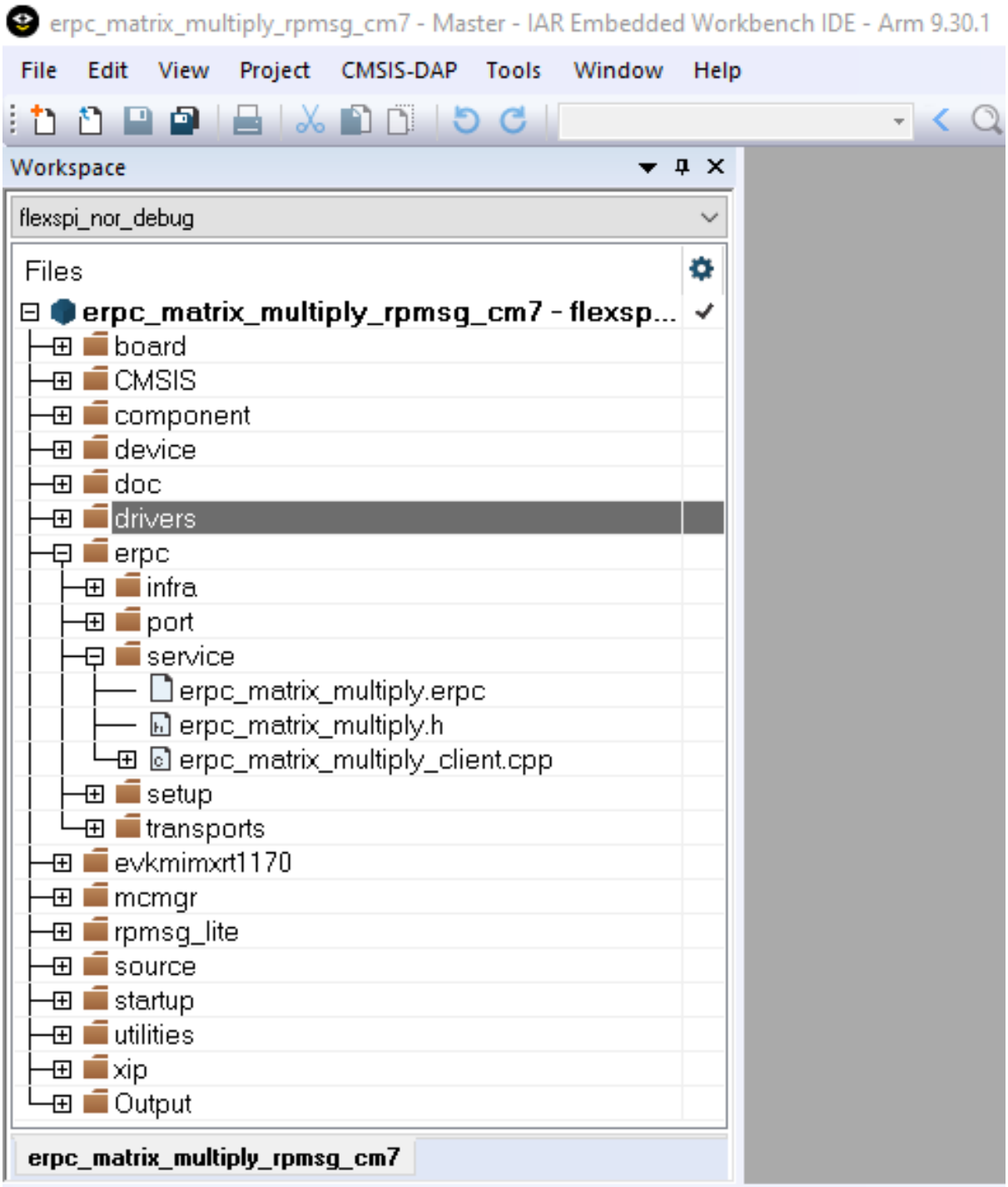
|

Parent topic: Multicore client application

Client-related generated files The client-related generated files are:

- erpc_matric_multiply.h
- erpc_matrix_multiply_client.cpp

These files contain the shim code for the functions and data types declared in the IDL file. These functions also call methods for codec initialization, data serialization, performing eRPC requests, and de-serializing outputs into expected data structures (if return values are expected). These shim code files can be found in the *<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service/* folder.



Parent topic: Multicore client application

Client infrastructure files The eRPC infrastructure files are located in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.

- Two files, erpc_client_manager.h and erpc_client_manager.cpp, are used for managing the client-side application. The main purpose of the client files is to create, perform, and release eRPC requests.
- Three files (erpc_codec.hpp, erpc_basic_codec.hpp, and erpc_basic_codec.cpp) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
- erpc_common.h file is used for common eRPC definitions, typedefs, and enums.
- erpc_manually_constructed.hpp file is used for allocating static storage for the used objects.
- Message buffer files are used for storing serialized data: erpc_message_buffer.hpp and erpc_message_buffer.cpp.
- erpc_transport.hpp file defines the abstract interface for transport layer.

The **port** subfolder contains the eRPC porting layer to adapt to different environments.

- erpc_port.h file contains definition of erpc_malloc() and erpc_free() functions.
- erpc_port_stdlib.cpp file ensures adaptation to stdlib.
- erpc_config_internal.h internal eRPC configuration file.

The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.

- erpc_client_setup.h and erpc_client_setup.cpp files needs to be added into the "Matrix multiply" example project to demonstrate the use of C-wrapped functions in this example.
- erpc_transport_setup.h and erpc_setup_rpmsg_lite_master.cpp files needs to be added into the project in order to allow C-wrapped function for transport layer setup.
- erpc_mbf_setup.h and erpc_setup_mbf_rpmsg.cpp files needs to be added into the project in order to allow message buffer factory usage.

The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions, in the setup folder.

- RPMsg-Lite is used as the transport layer for the communication between cores, erpc_rpmsg_lite_base_transport.hpp, erpc_rpmsg_lite_transport.hpp, and erpc_rpmsg_lite_transport.cpp files needs to be added into the client project.

erpc_matrix_multiply_rpmsg_cm7 - Master - IAR Embedded Workbench IDE - Arm 9.30.1
File Edit View Project CMSIS-DAP Tools Window Help
Workspace
flexspi_nor_debug
Files
erpc_matrix_multiply_rpmsg_cm7 - flexspi_n...
board
CMSIS
component
device
doc
drivers
erpc
infra
erpc_basic_codec.cpp
erpc_basic_codec.hpp
erpc_client_manager.cpp
erpc_client_manager.h
erpc_client_server_common.hpp
erpc_codec.hpp
erpc_common.h
erpc_crc16.cpp
erpc_crc16.hpp
erpc_manually_constructed.hpp
erpc_message_buffer.cpp
erpc_message_buffer.hpp
erpc_message_loggers.cpp
erpc_message_loggers.hpp
erpc_pre_post_action.cpp
erpc_pre_post_action.h
erpc_static_queue.hpp
erpc_transport.hpp
erpc_version.h
port
erpc_config_internal.h
erpc_endianness_agnostic_example.h
erpc_endianness_undefined.h
erpc_port.h
erpc_port_stdlib.cpp
erpc_setup_extensions.h
service
setup
erpc_client_setup.cpp
erpc_client_setup.h
erpc_mbf_setup.h
erpc_setup_mbf_rpmsg.cpp
erpc_setup_rpmsg_lite_master.cpp
erpc_transport_setup.h
transports
erpc_rpmsg_lite_base_transport.hpp
erpc_rpmsg_lite_transport.cpp
erpc_rpmsg_lite_transport.hpp
evkmimxrt1170
mcmgr
rpmsg_lite
source
startup
utilities
erpc_matrix_multiply_rpmsg_cm7

|

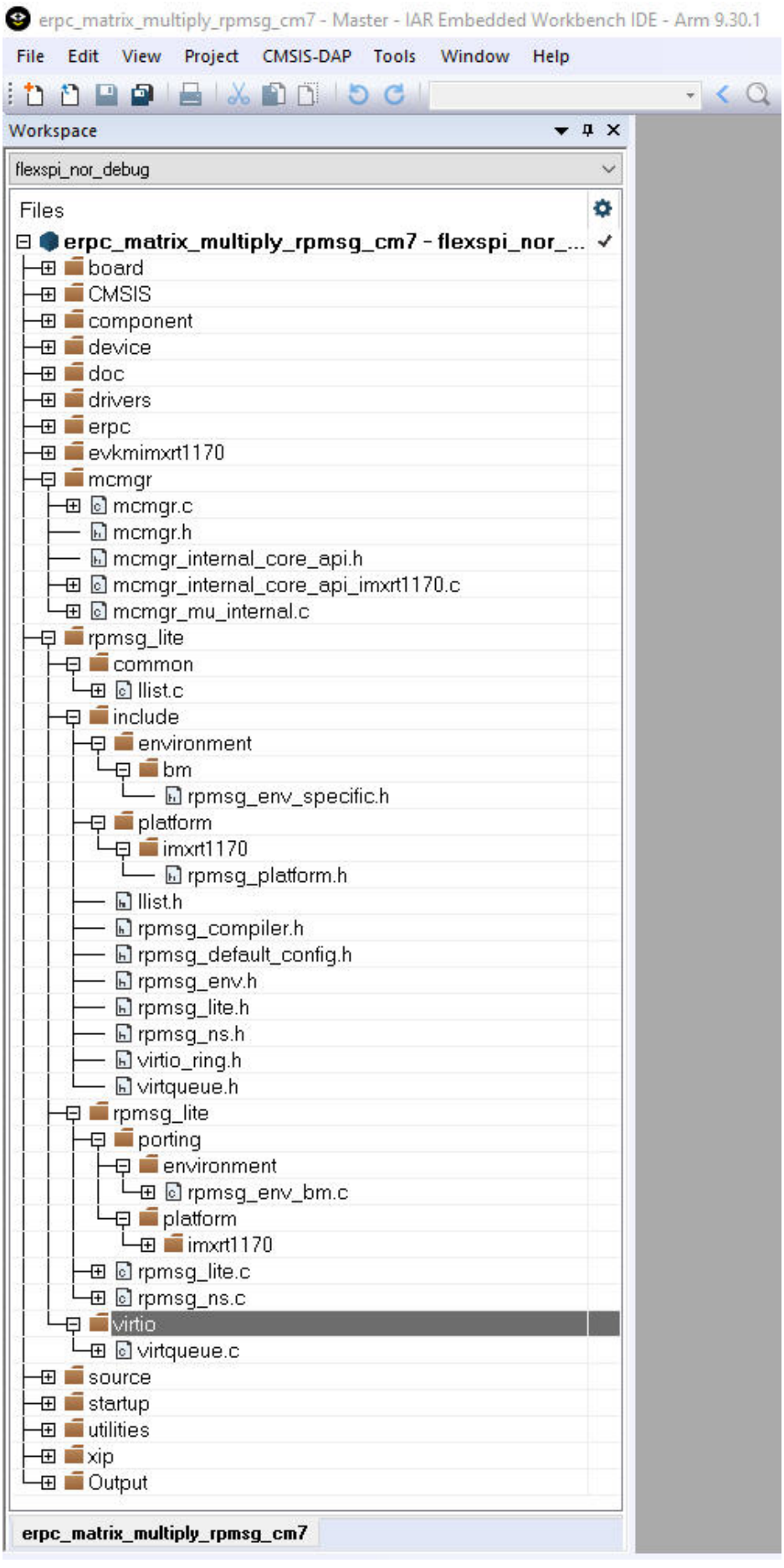
Parent topic: Multicore client application

Client multicore infrastructure files Because of the RPMsg-Lite (transport layer), it is also necessary to include RPMsg-Lite related files, which are in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/


erpc_matrix_multiply_rpmsg_cm7 - Master - IAR Embedded Workbench IDE - Arm 9.30.1
File
Edit
View
Project
CMSIS-DAP
Tools
Window
Help
Workspace
flexspi_nor_debug
Files
erpc_matrix_multiply_rpmsg_cm7 - flexspi_nor_...
board
CMSIS
component
device
doc
drivers
erpc
evkmimxrt1170
mcmgr
mcmgr.c
mcmgr.h
mcmgr_internal_core_api.h
mcmgr_internal_core_api_imxrt1170.c
mcmgr_mu_internal.c
rpmsg_lite
common
llist.c
include
environment
bm
rpmsg_env_specific.h
platform
imxrt1170
rpmsg_platform.h
llist.h
rpmsg_compiler.h
rpmsg_default_config.h
rpmsg_env.h
rpmsg_lite.h
rpmsg_ns.h
virtio_ring.h
virtqueue.h
rpmsg_lite
porting
environment
rpmsg_env_bm.c
platform
imxrt1170
rpmsg_lite.c
rpmsg_ns.c
virtio
virtqueue.c
source
startup
utilities
xip
Output
erpc_matrix_multiply_rpmsg_cm7

|

Parent topic: Multicore client application

Client user code The client's user code is stored in the main_core0.c file, located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_example/erpc_matrix_multiply_rpmsg/cm7

The main_core0.c file contains the code for target board and eRPC initialization.

- After initialization, the secondary core is released from reset.
- When the secondary core is ready, the primary core initializes two matrix variables.
- The erpcMatrixMultiply eRPC function is called to issue the eRPC request and get the result.

It is possible to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in erpc_error_handler.h and erpc_error_handler.cpp files.

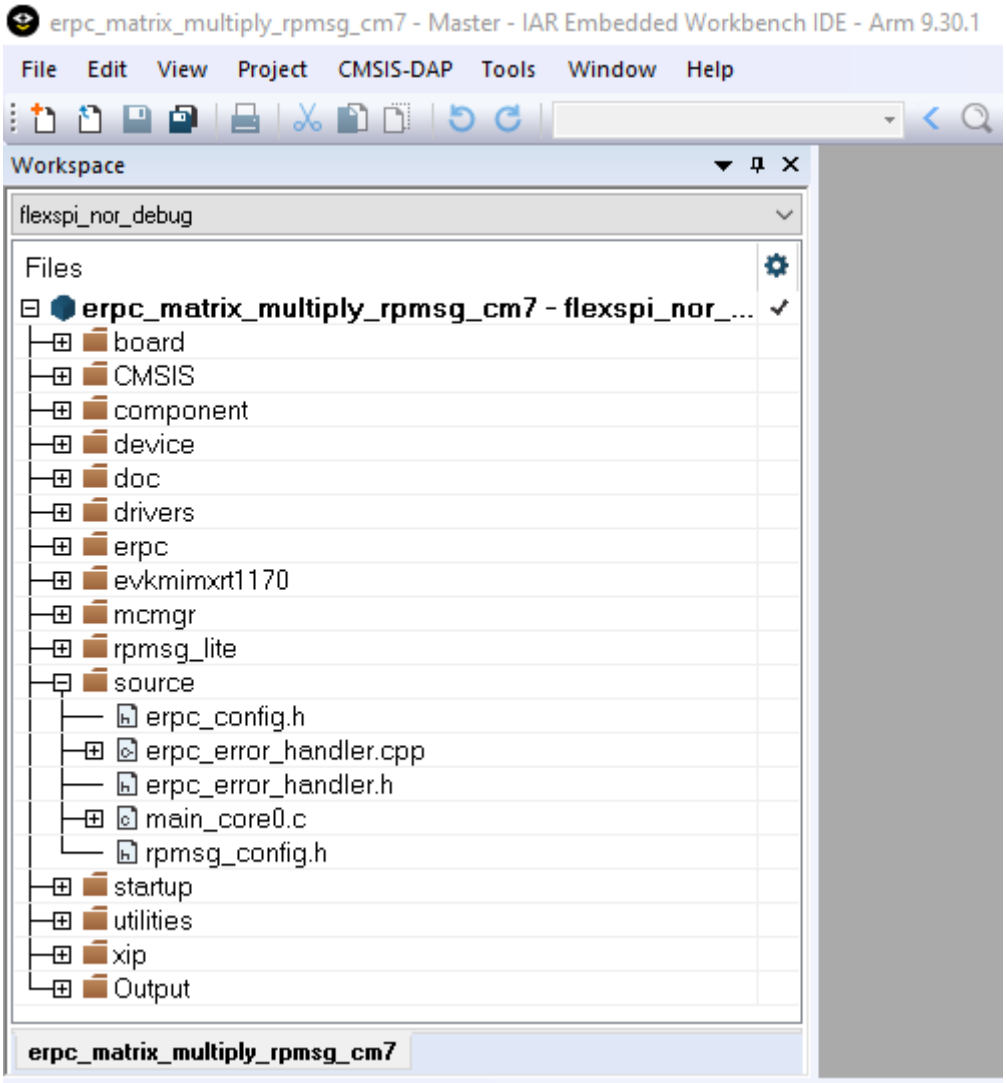
The matrix multiplication can be issued repeatedly, when pressing a software board button.

The eRPC-relevant code is captured in the following code snippet:

```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* RPMsg-Lite transport layer initialization */
erpc_transport_t transport;
transport = erpc_transport_rpmsg_lite_master_init(src, dst,
ERPC_TRANSPORT_RPMSG_LITE_LINK_ID);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_rpmsg_init(transport);
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport, message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
  /* Invoke the erpcMatrixMultiply function */
  erpcMatrixMultiply(matrix1, matrix2, result_matrix);
  ...
  /* Check if some error occured in eRPC */
  if (g_erpc_error_occurred)
  {
    /* Exit program loop */
    break;
  }
  ...
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg



|

|

Parent topic: Multicore client application

Parent topic:*Create an eRPC application*

Multiprocessor server application The "Matrix multiply" eRPC server project for multiprocessor applications is located in the *<MCUXpressoSDK_install_dir»/boards/<board_name>/multiprocessor_examples/ erpc_server_matrix_multiply_<transport_layer>* folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires server-related generated files (server shim code), server infrastructure files, and the server user code. There is no need for server multicore infrastructure files (MCMGR and RPMsg-Lite). The RPMsg-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

|SPI|<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_slave.cpp

<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.hpp

<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.cpp

| |UART|<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp

<eRPC base directory>/erpc_c/transports/erpc_uart_cmsis_transport.hpp

<eRPC base directory>/erpc_c/transports/erpc_uart_cmsis_transport.cpp

|

Server user code The server's user code is stored in the main_server.c file, located in the *<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>/* folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(Matrix matrix1, Matrix matrix2, Matrix result_matrix)
{
  ...
}
int main()
{
  ...
  /* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver␣
↪operations */
  erpc_transport_t transport;
  transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
  ...
  /* MessageBufferFactory initialization */
  erpc_mbf_t message_buffer_factory;
  message_buffer_factory = erpc_mbf_dynamic_init();
  ...
  /* eRPC server side initialization */
  erpc_server_t server;
  server = erpc_server_init(transport, message_buffer_factory);
  ...
  /* Adding the service to the server */
  erpc_service_t service = create_MatrixMultiplyService_service();
  erpc_add_service_to_server(server, service);
  ...
  while (1)
  {
    /* Process eRPC requests */
    erpc_status_t status = erpc_server_poll(server)
    /* handle error status */
    if (status != kErpcStatus_Success)
    {
      /* print error description */
      erpc_error_handler(status, 0);
      ...
    }
    ...
  }
}
```

Parent topic: Multiprocessor server application

Multiprocessor client application The "Matrix multiply" eRPC client project for multiprocessor applications is located in the *<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/iar/* folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires client-related generated files (server shim code),

client infrastructure files, and the client user code. There is no need for client multicore infrastructure files (MCMGR and RPMsg-Lite). The RPMsg-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

|SPI|<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_master.cpp

<eRPC base directory>/erpc_c/transports/ erpc_(d)spi_master_transport.hpp

<eRPC base directory>/erpc_c/transports/ erpc_(d)spi_master_transport.cpp

| |UART|<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp

<eRPC base directory>/erpc_c/transports/erpc_uart_cmsis_transport.hpp

<eRPC base directory>/erpc_c/transports/erpc_uart_cmsis_transport.cpp

|

Client user code The client's user code is stored in the main_client.c file, located in the *<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/* folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

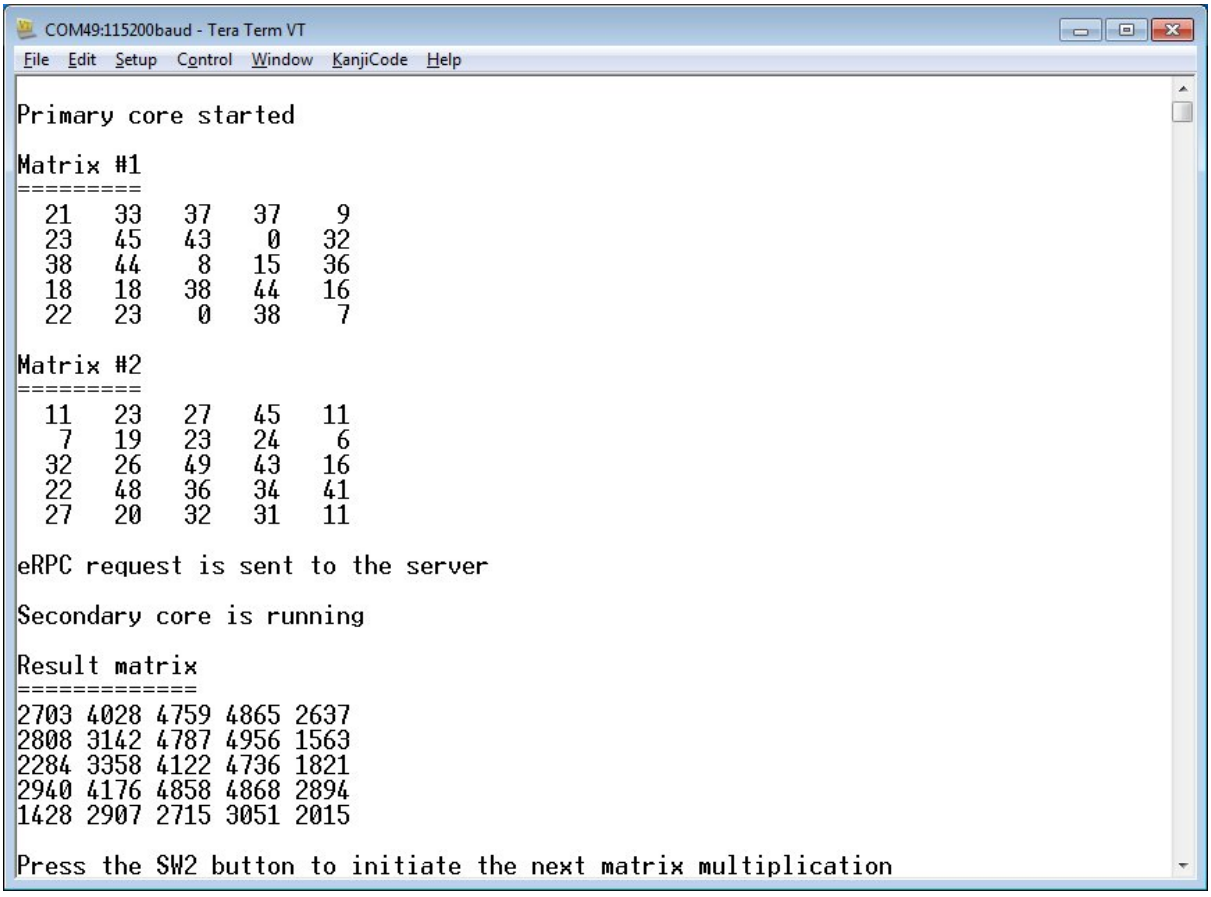
```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver␣
↪operations */
erpc_transport_t transport;
transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_dynamic_init();
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport,message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
  /* Invoke the erpcMatrixMultiply function */
  erpcMatrixMultiply(matrix1, matrix2, result_matrix);
  ...
  /* Check if some error occured in eRPC */
  if (g_erpc_error_occurred)
  {
    /* Exit program loop */
    break;
  }
  ...
}
```

Parent topic: Multiprocessor client application

Parent topic: Multiprocessor server application

Parent topic:*Create an eRPC application*

Running the eRPC application Follow the instructions in *Getting Started with MCUXpresso SDK* (document MCUXSDKGSUG) (located in the *<MCUXpressoSDK_install_dir>/docs* folder), to load both the primary and the secondary core images into the on-chip memory, and then effectively debug the dual-core application. After the application is running, the serial console should look like:



For multiprocessor applications that are running between PC and the target evaluation board or between two boards, follow the instructions in the accompanied example readme files that provide details about the proper board setup and the PC side setup (Python).

Parent topic:*Create an eRPC application*

Parent topic:*eRPC example*

eRPC example This section shows how to create an example eRPC application called "Matrix multiply", which implements one eRPC function (matrix multiply) with two function parameters (two matrices). The client-side application calls this eRPC function, and the server side performs the multiplication of received matrices. The server side then returns the result.

For example, use the NXP MIMXRT1170-EVK board as the target dual-core platform, and the IAR Embedded Workbench for ARM (EWARM) as the target IDE for developing the eRPC example.

- The primary core (CM7) runs the eRPC client.
- The secondary core (CM4) runs the eRPC server.
- RPMsg-Lite (Remote Processor Messaging Lite) is used as the eRPC transport layer.

The "Matrix multiply" application can be also run in the multi-processor setup. In other words, the eRPC client running on one SoC comunicates with the eRPC server that runs on anothe SoC, utilizing different transport channels. It is possible to run the board-to-PC example (PC as the eRPC server and a board as the eRPC client, and vice versa) and also the board-to-board example. These multiprocessor examples are prepared for selected boards only.

|Multicore application source and project files|*<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_*
|Multiprocessor application source and project files|*<MCUXpressoSDK_install_dir>/boards/<board_name>/multi*

<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<tr

| |eRPC source files|*<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/*| |RPMsg-Lite source files|*<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/*|

Designing the eRPC application The matrix multiply application is based on calling single eRPC function that takes 2 two-dimensional arrays as input and returns matrix multiplication results as another 2 two-dimensional array. The IDL file syntax supports arrays with the dimension length set by the number only (in the current eRPC implementation). Because of this, a variable is declared in the IDL dedicated to store information about matrix dimension length, and to allow easy maintenance of the user and server code.

For a simple use of the two-dimensional array, the alias name (new type definition) for this data type has is declared in the IDL. Declaring this alias name ensures that the same data type can be used across the client and server applications.

Parent topic:*eRPC example*

Creating the IDL file The created IDL file is located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/s

The created IDL file contains the following code:

```
program erpc_matrix_multiply
/*! This const defines the matrix size. The value has to be the same as the
Matrix array dimension. Do not forget to re-generate the erpc code once the
matrix size is changed in the erpc file */
const int32 matrix_size = 5;
/*! This is the matrix array type. The dimension has to be the same as the
matrix size const. Do not forget to re-generate the erpc code once the
matrix size is changed in the erpc file */
type Matrix = int32[matrix_size][matrix_size];
interface MatrixMultiplyService {
erpcMatrixMultiply(in Matrix matrix1, in Matrix matrix2, out Matrix result_matrix) ->
void
}
```

Details:

- The IDL file starts with the program name (*erpc_matrix_multiply*), and this program name is used in the naming of all generated outputs.
- The declaration and definition of the constant variable named *matrix_size* follows next. The *matrix_size* variable is used for passing information about the length of matrix dimensions to the client/server user code.
- The alias name for the two-dimensional array type (*Matrix*) is declared.
- The interface group *MatrixMultiplyService* is located at the end of the IDL file. This interface group contains only one function declaration *erpcMatrixMultiply*.
- As shown above, the function's declaration contains three parameters of Matrix type: *matrix1* and *matrix2* are input parameters, while *result_matrix* is the output parameter. Additionally, the returned data type is declared as void.

When writing the IDL file, the following order of items is recommended:

1. Program name at the top of the IDL file.
2. New data types and constants declarations.
3. Declarations of interfaces and functions at the end of the IDL file.

Parent topic:*eRPC example*

Using the eRPC generator tool

|Windows OS|<MCUXpressoSDK_install_dir>/middleware/multicore/tools/erp
|Linux OS|<MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Linux_x64

<MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Linux_x86

| |Mac OS|<MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Mac|

The files for the "Matrix multiply" example are pre-generated and already a part of the application projects. The following section describes how they have been created.

- The easiest way to create the shim code is to copy the erpcgen application to the same folder where the IDL file (*.erpc) is located; then run the following command:

  erpcgen <IDL_file>.erpc

- In the "Matrix multiply" example, the command should look like:

  erpcgen erpc_matrix_multiply.erpc

Additionally, another method to create the shim code is to execute the eRPC application using input commands:

- "-?"/"—help" – Shows supported commands.
- "-o <filePath>"/"—output<filePath>" – Sets the output directory.

For example,

```
<path_to_erpcgen>/erpcgen –o <path_to_output>
<path_to_IDL>/<IDL_file_name>.erpc
```

For the "Matrix multiply" example, when the command is executed from the default erpcgen location, it looks like:

erpcgen –o

../../../../../boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service
../../../../../boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service/erpc_matrix_mu

In both cases, the following four files are generated into the *<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service folder.*

- erpc_matrix_multiply.h
- erpc_matrix_multiply_client.cpp
- erpc_matrix_multiply_server.h
- erpc_matrix_multiply_server.cpp

For multiprocessor examples, the eRPC file and pre-generated files can be found in the *<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_common/erpc_matrix_multiply/serv* folder.

For Linux OS users:

- Do not forget to set the permissions for the eRPC generator application.
- Run the application as ./erpcgen... instead of as erpcgen ....

Parent topic:*eRPC example*

Create an eRPC application This section describes a generic way to create a client/server eRPC application:

1. **Design the eRPC application:** Decide which data types are sent between applications, and define functions that send/receive this data.
2. **Create the IDL file:** The IDL file contains information about data types and functions used in an eRPC application, and is written in the IDL language.
3. **Use the eRPC generator tool:** This tool takes an IDL file and generates the shim code for the client and the server-side applications.
4. **Create an eRPC application:**
    1. Create two projects, where one project is for the client side (primary core) and the other project is for the server side (secondary core).
    2. Add generated files for the client application to the client project, and add generated files for the server application to the server project.
    3. Add infrastructure files.
    4. Add user code for client and server applications.
    5. Set the client and server project options.
5. **Run the eRPC application:** Run both the server and the client applications. Make sure that the server has been run before the client request was sent.

A specific example follows in the next section.

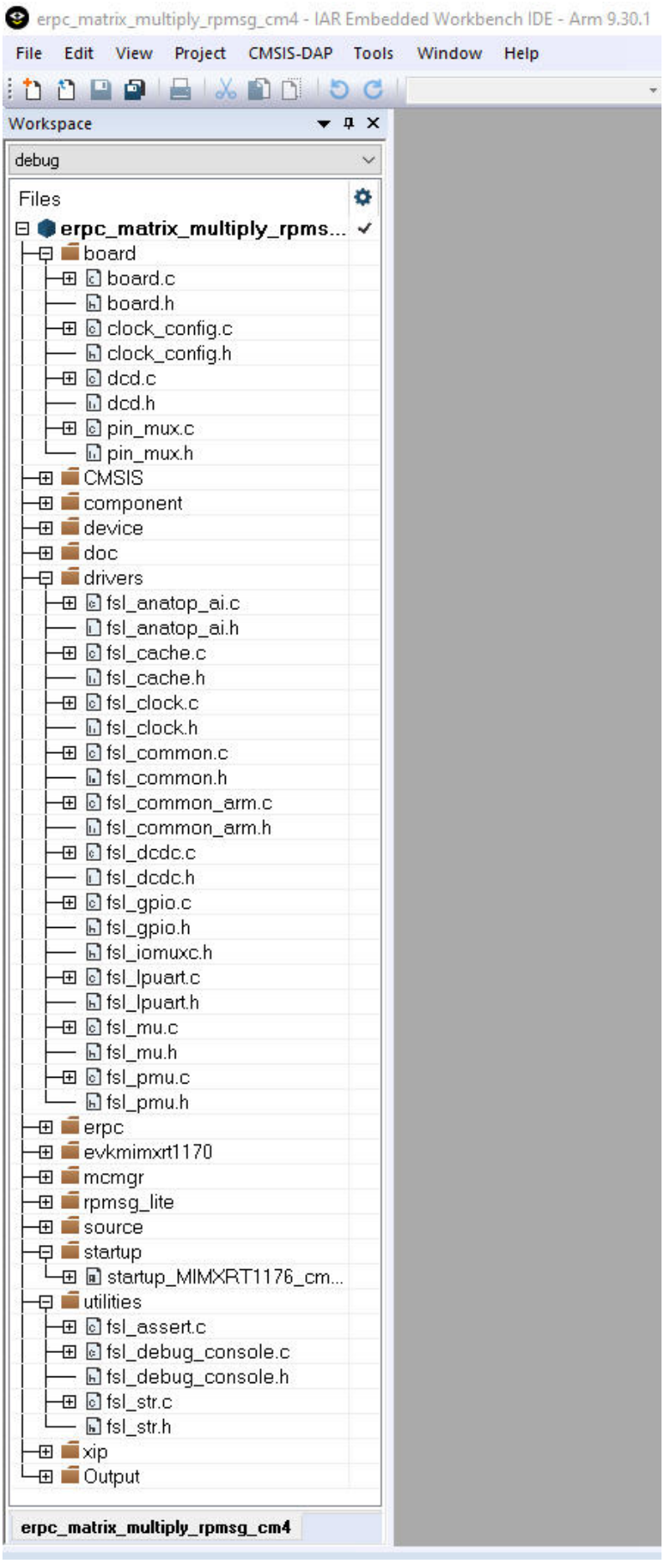
Multicore server application The "Matrix multiply" eRPC server project is located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm4/iar

The project files for the eRPC server have the `_cm4` suffix.

Server project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in:

- *<MCUXpressoSDK_install_dir>/devices/<device>*
- *<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/*


erpc_matrix_multiply_rpmsg_cm4 - IAR Embedded Workbench IDE - Arm 9.30.1
File Edit View Project CMSIS-DAP Tools Window Help
Workspace
debug
Files
erpc_matrix_multiply_rpms...
board
board.c
board.h
clock_config.c
clock_config.h
dcd.c
dcd.h
pin_mux.c
pin_mux.h
CMSIS
component
device
doc
drivers
fsl_anatop_ai.c
fsl_anatop_ai.h
fsl_cache.c
fsl_cache.h
fsl_clock.c
fsl_clock.h
fsl_common.c
fsl_common.h
fsl_common_arm.c
fsl_common_arm.h
fsl_dcdc.c
fsl_dcdc.h
fsl_gpio.c
fsl_gpio.h
fsl_iomuxc.h
fsl_lpuart.c
fsl_lpuart.h
fsl_mu.c
fsl_mu.h
fsl_pmu.c
fsl_pmu.h
erpc
evkmimxrt1170
mcmgr
rpmsg_lite
source
startup
startup_MIMXRT1176_cm...
utilities
fsl_assert.c
fsl_debug_console.c
fsl_debug_console.h
fsl_str.c
fsl_str.h
xip
Output
erpc_matrix_multiply_rpmsg_cm4

|

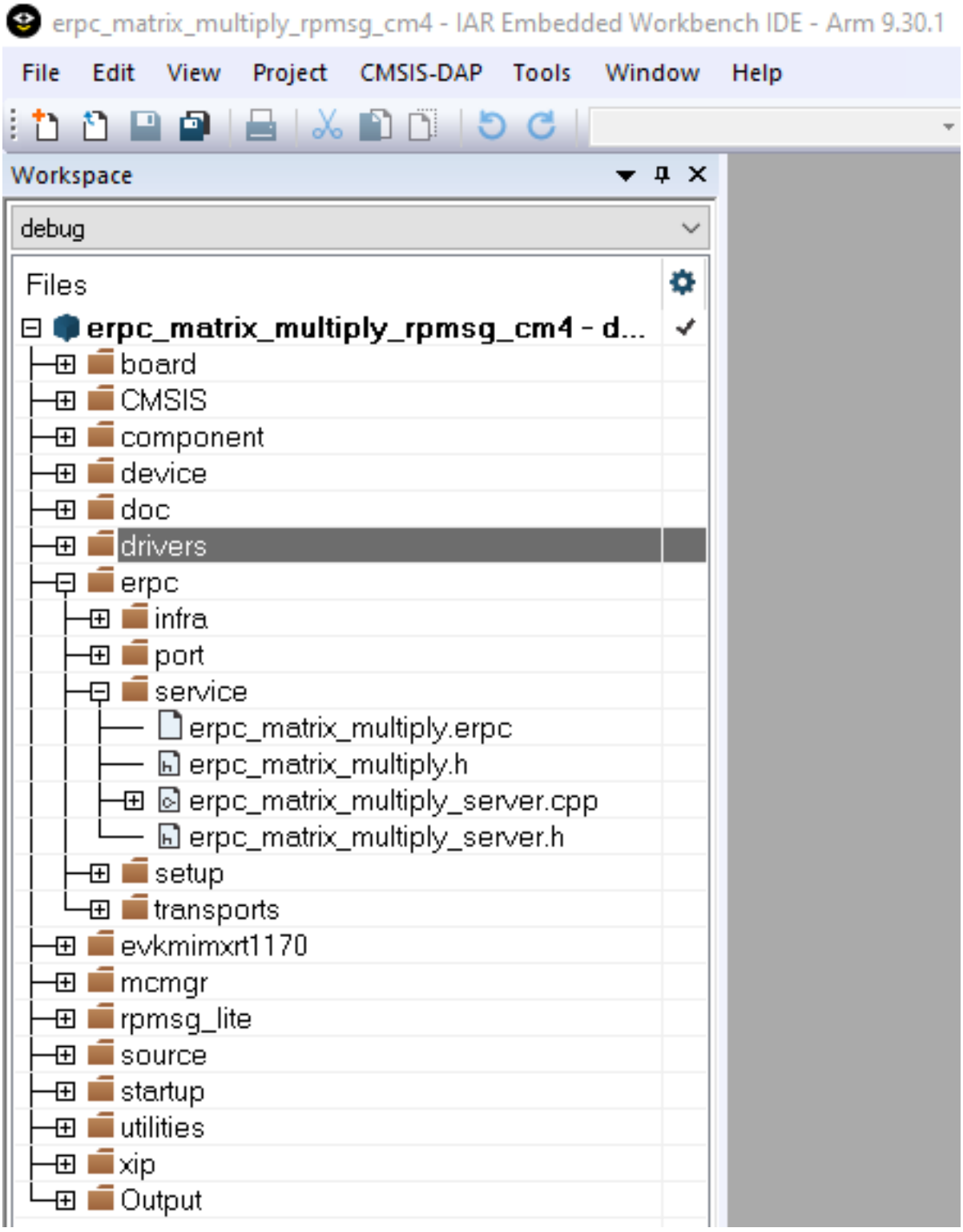
Parent topic: Multicore server application

Server related generated files The server-related generated files are:

- erpc_matric_multiply.h
- erpc_matrix_multiply_server.h
- erpc_matrix_multiply_server.cpp

The server-related generated files contain the shim code for functions and data types declared in the IDL file. These files also contain functions for the identification of client requested functions, data deserialization, calling requested function's implementations, and data serialization and return, if requested by the client. These shim code files can be found in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/s



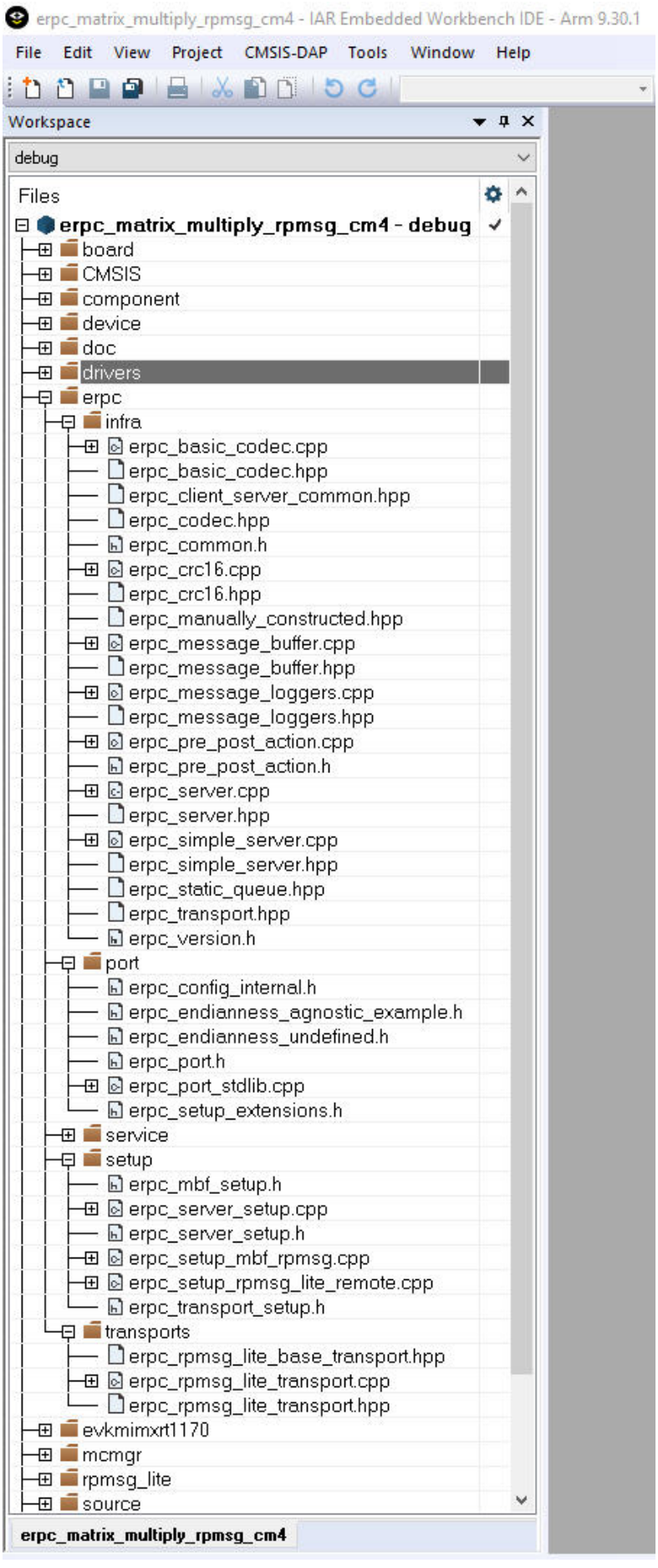
Parent topic: Multicore server application

Server infrastructure files The eRPC infrastructure files are located in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c

The **erpc_c** folder contains files for creating eRPC client and server applications in the *C/C++* language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.
  - Four files, erpc_server.hpp, erpc_server.cpp, erpc_simple_server.hpp, and erpc_simple_server.cpp, are used for running the eRPC server on the server-side applications. The simple server is currently the only implementation of the server, and its role is to catch client requests, identify and call requested functions, and send data back when requested.
  - Three files (erpc_codec.hpp, erpc_basic_codec.hpp, and erpc_basic_codec.cpp) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
  - The erpc_common.hpp file is used for common eRPC definitions, typedefs, and enums.
  - The erpc_manually_constructed.hpp file is used for allocating static storage for the used objects.
  - Message buffer files are used for storing serialized data: erpc_message_buffer.h and erpc_message_buffer.cpp.
  - The erpc_transport.h file defines the abstract interface for transport layer.
- The **port** subfolder contains the eRPC porting layer to adapt to different environments.
  - erpc_port.h file contains definition of erpc_malloc() and erpc_free() functions.
  - erpc_port_stdlib.cpp file ensures adaptation to stdlib.
  - erpc_config_internal.h internal erpc configuration file.
- The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.
  - The erpc_server_setup.h and erpc_server_setup.cpp files needs to be added into the "Matrix multiply" example project to demonstrate the use of C-wrapped functions in this example.
  - The erpc_transport_setup.h and erpc_setup_rpmsg_lite_remote.cpp files needs to be added into the project in order to allow the C-wrapped function for transport layer setup.
  - The erpc_mbf_setup.h and erpc_setup_mbf_rpmsg.cpp files needs to be added into the project in order to allow message buffer factory usage.
- The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions in the setup folder.
  - RPMsg-Lite is used as the transport layer for the communication between cores, erpc_rpmsg_lite_base_transport.hpp, erpc_rpmsg_lite_transport.hpp, and erpc_rpmsg_lite_transport.cpp files need to be added into the server project.


erpc_matrix_multiply_rpmsg_cm4 - IAR Embedded Workbench IDE - Arm 9.30.1
File Edit View Project CMSIS-DAP Tools Window Help
Workspace
debug
Files
erpc_matrix_multiply_rpmsg_cm4 - debug
board
CMSIS
component
device
doc
drivers
erpc
infra
erpc_basic_codec.cpp
erpc_basic_codec.hpp
erpc_client_server_common.hpp
erpc_codec.hpp
erpc_common.h
erpc_crc16.cpp
erpc_crc16.hpp
erpc_manually_constructed.hpp
erpc_message_buffer.cpp
erpc_message_buffer.hpp
erpc_message_loggers.cpp
erpc_message_loggers.hpp
erpc_pre_post_action.cpp
erpc_pre_post_action.h
erpc_server.cpp
erpc_server.hpp
erpc_simple_server.cpp
erpc_simple_server.hpp
erpc_static_queue.hpp
erpc_transport.hpp
erpc_version.h
port
erpc_config_internal.h
erpc_endianness_agnostic_example.h
erpc_endianness_undefined.h
erpc_port.h
erpc_port_stdlib.cpp
erpc_setup_extensions.h
service
setup
erpc_mbf_setup.h
erpc_server_setup.cpp
erpc_server_setup.h
erpc_setup_mbf_rpmsg.cpp
erpc_setup_rpmsg_lite_remote.cpp
erpc_transport_setup.h
transports
erpc_rpmsg_lite_base_transport.hpp
erpc_rpmsg_lite_transport.cpp
erpc_rpmsg_lite_transport.hpp
evkmimxrt1170
mcmgr
rpmsg_lite
source
erpc_matrix_multiply_rpmsg_cm4

|

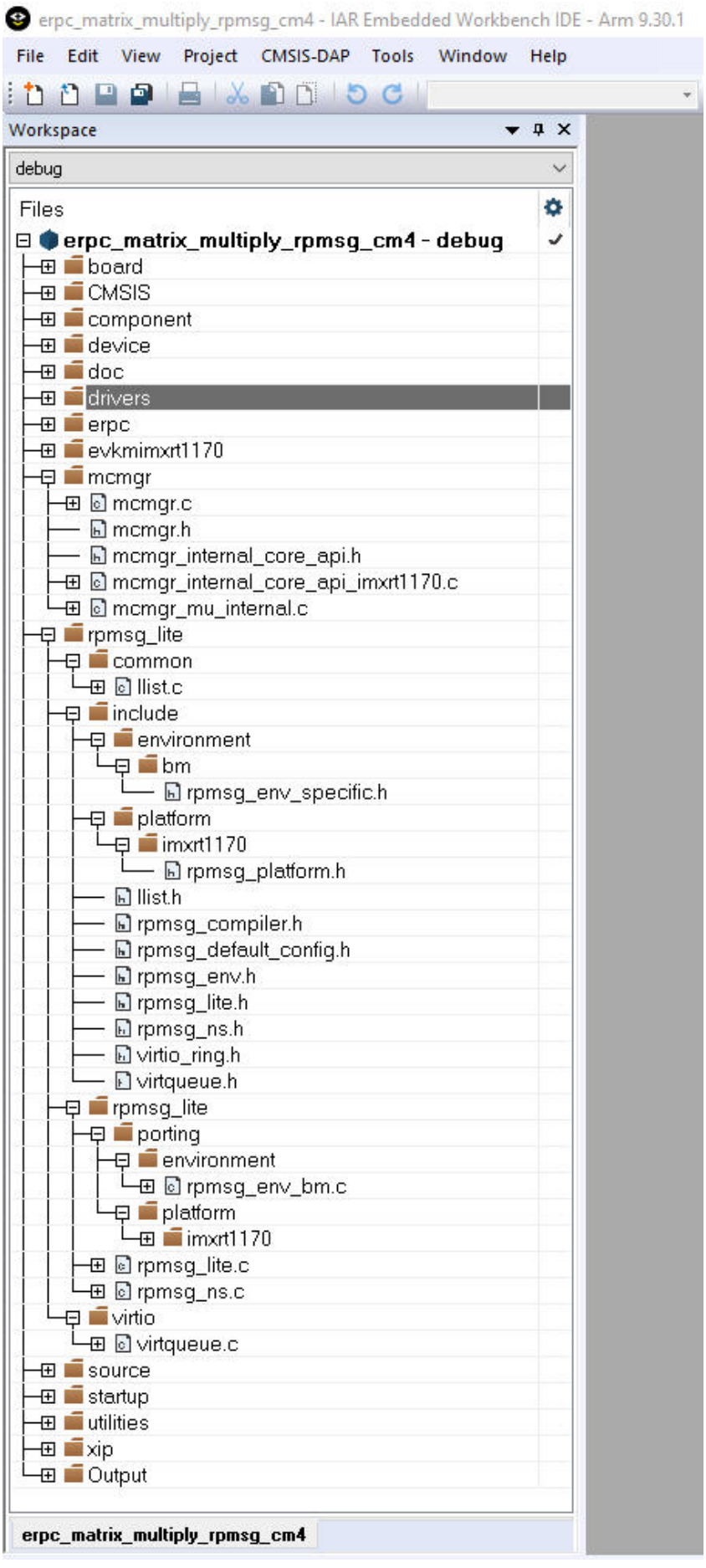
Parent topic: Multicore server application

Server multicore infrastructure files Because of the RPMsg-Lite (transport layer), it is also necessary to include RPMsg-Lite related files, which are in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/


erpc_matrix_multiply_rpmsg_cm4 - IAR Embedded Workbench IDE - Arm 9.30.1
File Edit View Project CMSIS-DAP Tools Window Help
Workspace
debug
Files
erpc_matrix_multiply_rpmsg_cm4 - debug
board
CMSIS
component
device
doc
drivers
erpc
evkmimxrt1170
mcmgr
mcmgr.c
mcmgr.h
mcmgr_internal_core_api.h
mcmgr_internal_core_api_imxrt1170.c
mcmgr_mu_internal.c
rpmsg_lite
common
llist.c
include
environment
bm
rpmsg_env_specific.h
platform
imxrt1170
rpmsg_platform.h
llist.h
rpmsg_compiler.h
rpmsg_default_config.h
rpmsg_env.h
rpmsg_lite.h
rpmsg_ns.h
virtio_ring.h
virtqueue.h
rpmsg_lite
porting
environment
rpmsg_env_bm.c
platform
imxrt1170
rpmsg_lite.c
rpmsg_ns.c
virtio
virtqueue.c
source
startup
utilities
xip
Output
erpc_matrix_multiply_rpmsg_cm4

|

Parent topic: Multicore server application

Server user code The server's user code is stored in the main_core1.c file, located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm4

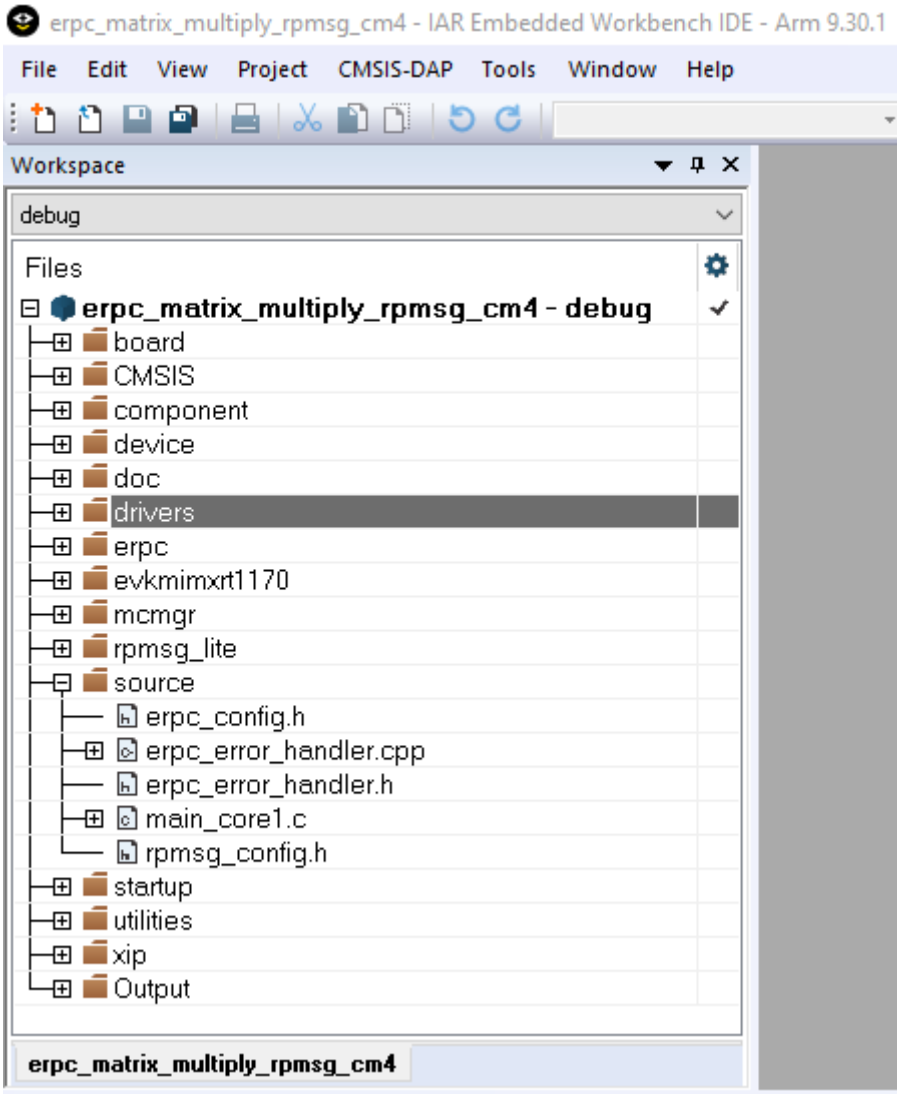
The main_core1.c file contains two functions:

- The **main()** function contains the code for the target board and eRPC server initialization. After the initialization, the matrix multiply service is added and the eRPC server waits for client's requests in the while loop.
- The **erpcMatrixMultiply()** function is the user implementation of the eRPC function defined in the IDL file.
- There is the possibility to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in the erpc_error_handler.h and erpc_error_handler.cpp files.

The eRPC-relevant code is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(const Matrix *matrix1, const Matrix *matrix2, Matrix *result_matrix)
{
  ...
}
int main()
{
  ...
  /* RPMsg-Lite transport layer initialization */
  erpc_transport_t transport;
  transport = erpc_transport_rpmsg_lite_remote_init(src, dst, (void*)startupData,
  ERPC_TRANSPORT_RPMSG_LITE_LINK_ID, SignalReady, NULL);
  ...
  /* MessageBufferFactory initialization */
  erpc_mbf_t message_buffer_factory;
  message_buffer_factory = erpc_mbf_rpmsg_init(transport);
  ...
  /* eRPC server side initialization */
  erpc_server_t server;
  server = erpc_server_init(transport, message_buffer_factory);
  ...
  /* Adding the service to the server */
  erpc_service_t service = create_MatrixMultiplyService_service();
  erpc_add_service_to_server(server, service);
  ...
  while (1)
  {
    /* Process eRPC requests */
    erpc_status_t status = erpc_server_poll(server);
    /* handle error status */
    if (status != kErpcStatus_Success)
    {
      /* print error description */
      erpc_error_handler(status, 0);
      ...
    }
    ...
  }
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the *<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/ erpc_matrix_multiply_rpmsg* folder.



Parent topic: Multicore server application

Parent topic:*Create an eRPC application*

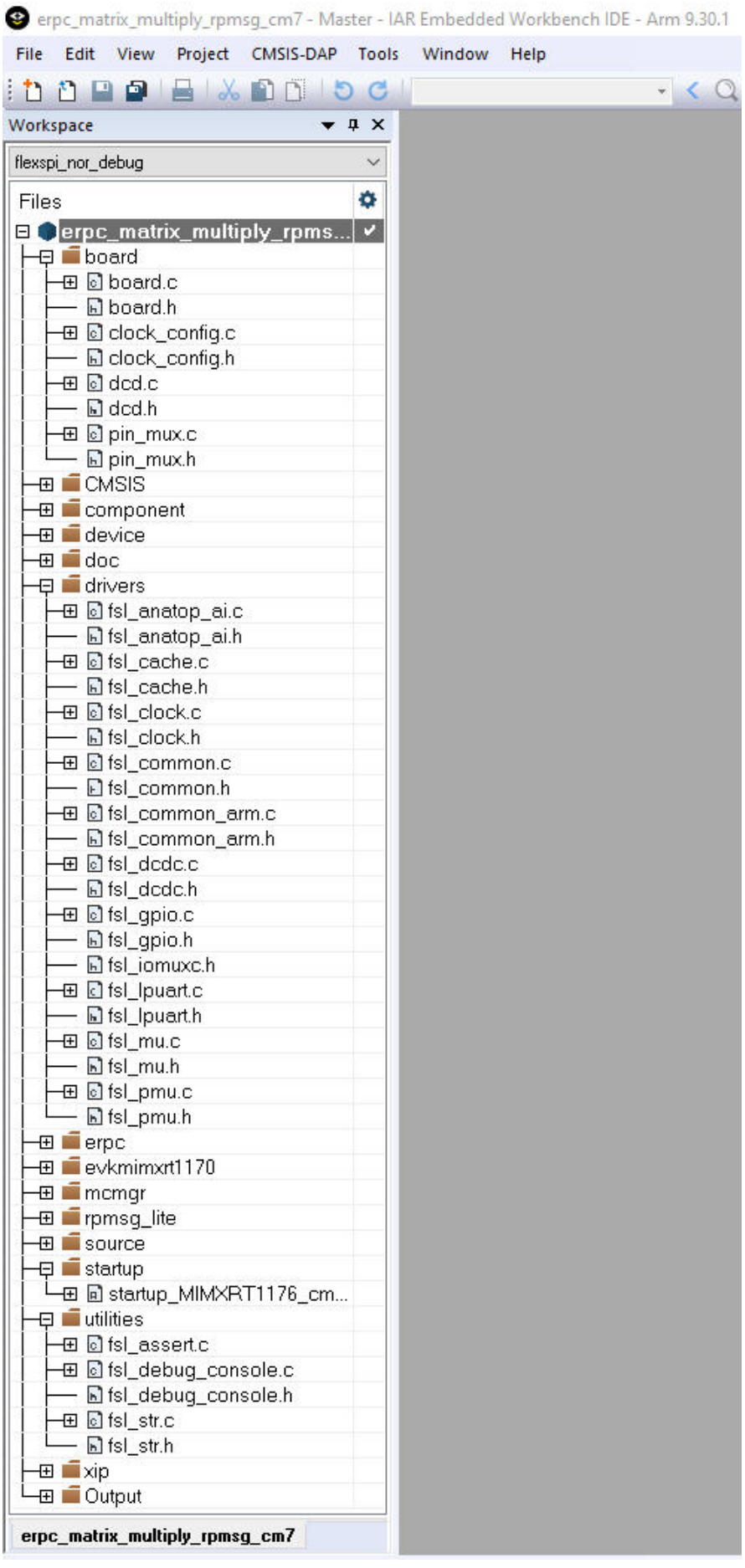
Multicore client application The "Matrix multiply" eRPC client project is located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm7/iar

Project files for the eRPC client have the _cm7 suffix.

Client project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in the following folders:

- *<MCUXpressoSDK_install_dir>/devices/<device>*
- *<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/*


erpc_matrix_multiply_rpmsg_cm7 - Master - IAR Embedded Workbench IDE - Arm 9.30.1
File
Edit
View
Project
CMSIS-DAP
Tools
Window
Help
Workspace
flexspi_nor_debug
Files
erpc_matrix_multiply_rpms...
board
board.c
board.h
clock_config.c
clock_config.h
dcd.c
dcd.h
pin_mux.c
pin_mux.h
CMSIS
component
device
doc
drivers
fsl_anatop_ai.c
fsl_anatop_ai.h
fsl_cache.c
fsl_cache.h
fsl_clock.c
fsl_clock.h
fsl_common.c
fsl_common.h
fsl_common_arm.c
fsl_common_arm.h
fsl_dcdc.c
fsl_dcdc.h
fsl_gpio.c
fsl_gpio.h
fsl_iomuxc.h
fsl_lpuart.c
fsl_lpuart.h
fsl_mu.c
fsl_mu.h
fsl_pmu.c
fsl_pmu.h
erpc
evkmimxrt1170
mcmgr
rpmsg_lite
source
startup
startup_MIMXRT1176_cm...
utilities
fsl_assert.c
fsl_debug_console.c
fsl_debug_console.h
fsl_str.c
fsl_str.h
xip
Output
erpc_matrix_multiply_rpmsg_cm7

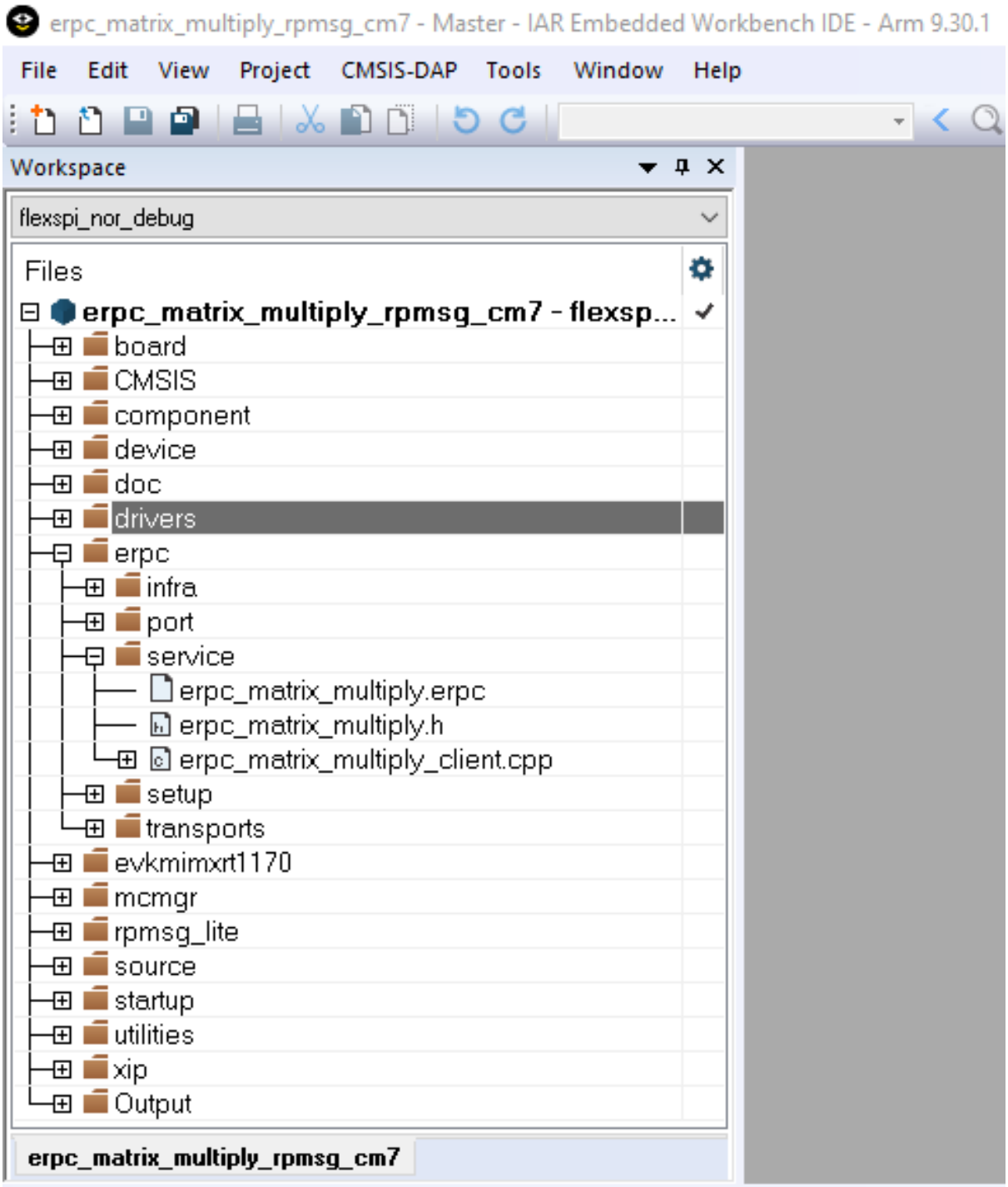
|

Parent topic: Multicore client application

Client-related generated files The client-related generated files are:

- erpc_matric_multiply.h
- erpc_matrix_multiply_client.cpp

These files contain the shim code for the functions and data types declared in the IDL file. These functions also call methods for codec initialization, data serialization, performing eRPC requests, and de-serializing outputs into expected data structures (if return values are expected). These shim code files can be found in the *<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service/* folder.



Parent topic: Multicore client application

Client infrastructure files The eRPC infrastructure files are located in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.

- Two files, erpc_client_manager.h and erpc_client_manager.cpp, are used for managing the client-side application. The main purpose of the client files is to create, perform, and release eRPC requests.
- Three files (erpc_codec.hpp, erpc_basic_codec.hpp, and erpc_basic_codec.cpp) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
- erpc_common.h file is used for common eRPC definitions, typedefs, and enums.
- erpc_manually_constructed.hpp file is used for allocating static storage for the used objects.
- Message buffer files are used for storing serialized data: erpc_message_buffer.hpp and erpc_message_buffer.cpp.
- erpc_transport.hpp file defines the abstract interface for transport layer.

The **port** subfolder contains the eRPC porting layer to adapt to different environments.

- erpc_port.h file contains definition of erpc_malloc() and erpc_free() functions.
- erpc_port_stdlib.cpp file ensures adaptation to stdlib.
- erpc_config_internal.h internal eRPC configuration file.

The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.

- erpc_client_setup.h and erpc_client_setup.cpp files needs to be added into the "Matrix multiply" example project to demonstrate the use of C-wrapped functions in this example.
- erpc_transport_setup.h and erpc_setup_rpmsg_lite_master.cpp files needs to be added into the project in order to allow C-wrapped function for transport layer setup.
- erpc_mbf_setup.h and erpc_setup_mbf_rpmsg.cpp files needs to be added into the project in order to allow message buffer factory usage.

The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions, in the setup folder.

- RPMsg-Lite is used as the transport layer for the communication between cores, erpc_rpmsg_lite_base_transport.hpp, erpc_rpmsg_lite_transport.hpp, and erpc_rpmsg_lite_transport.cpp files needs to be added into the client project.

erpc_matrix_multiply_rpmsg_cm7 - Master - IAR Embedded Workbench IDE - Arm 9.30.1
File Edit View Project CMSIS-DAP Tools Window Help
Workspace
flexspi_nor_debug
Files
erpc_matrix_multiply_rpmsg_cm7 - flexspi_n...
board
CMSIS
component
device
doc
drivers
erpc
infra
erpc_basic_codec.cpp
erpc_basic_codec.hpp
erpc_client_manager.cpp
erpc_client_manager.h
erpc_client_server_common.hpp
erpc_codec.hpp
erpc_common.h
erpc_crc16.cpp
erpc_crc16.hpp
erpc_manually_constructed.hpp
erpc_message_buffer.cpp
erpc_message_buffer.hpp
erpc_message_loggers.cpp
erpc_message_loggers.hpp
erpc_pre_post_action.cpp
erpc_pre_post_action.h
erpc_static_queue.hpp
erpc_transport.hpp
erpc_version.h
port
erpc_config_internal.h
erpc_endianness_agnostic_example.h
erpc_endianness_undefined.h
erpc_port.h
erpc_port_stdlib.cpp
erpc_setup_extensions.h
service
setup
erpc_client_setup.cpp
erpc_client_setup.h
erpc_mbf_setup.h
erpc_setup_mbf_rpmsg.cpp
erpc_setup_rpmsg_lite_master.cpp
erpc_transport_setup.h
transports
erpc_rpmsg_lite_base_transport.hpp
erpc_rpmsg_lite_transport.cpp
erpc_rpmsg_lite_transport.hpp
evkmimxrt1170
mcmgr
rpmsg_lite
source
startup
utilities
erpc_matrix_multiply_rpmsg_cm7

|

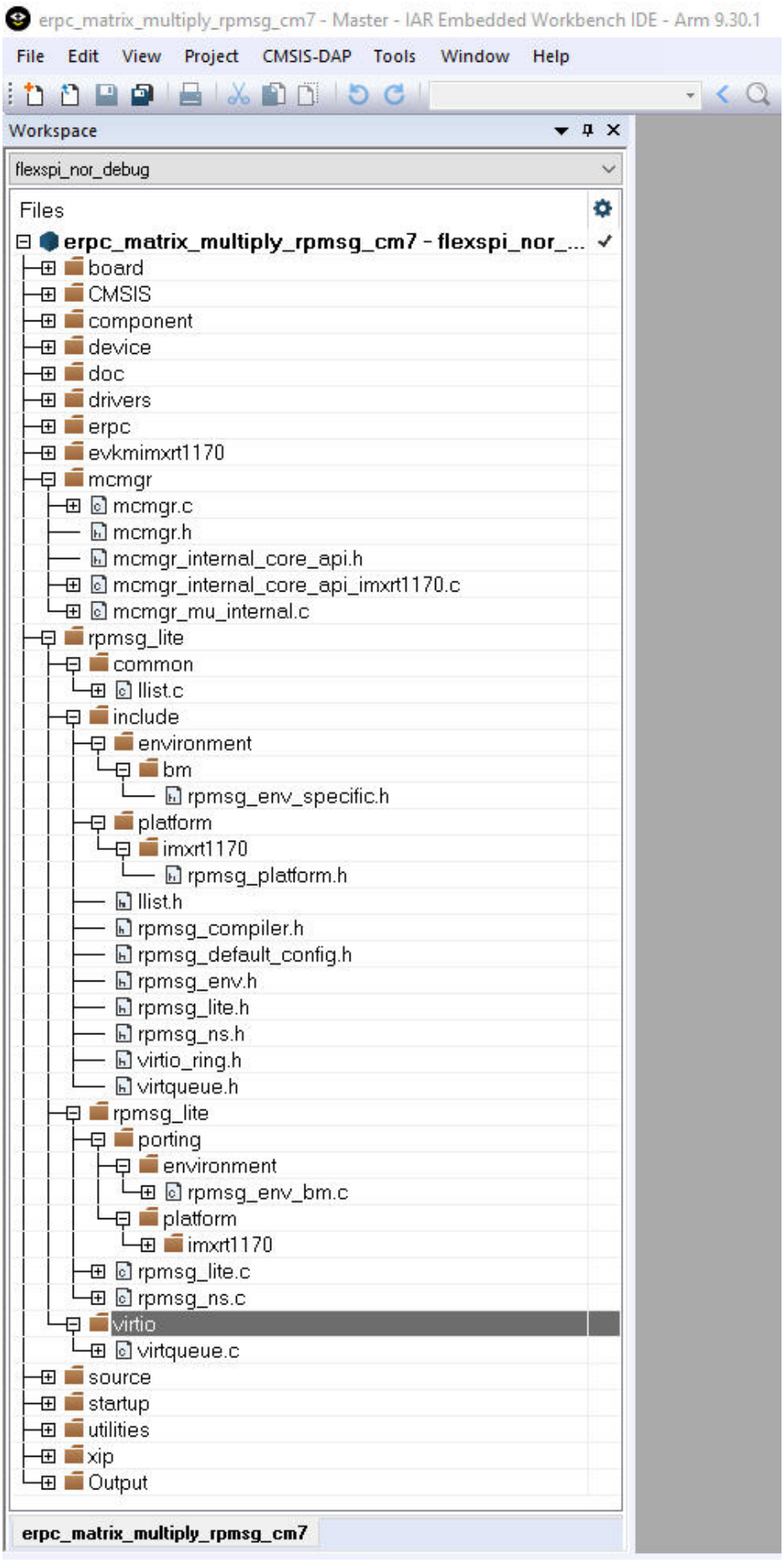
Parent topic: Multicore client application

Client multicore infrastructure files Because of the RPMsg-Lite (transport layer), it is also necessary to include RPMsg-Lite related files, which are in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/


erpc_matrix_multiply_rpmsg_cm7 - Master - IAR Embedded Workbench IDE - Arm 9.30.1
File
Edit
View
Project
CMSIS-DAP
Tools
Window
Help
Workspace
flexspi_nor_debug
Files
erpc_matrix_multiply_rpmsg_cm7 - flexspi_nor_...
board
CMSIS
component
device
doc
drivers
erpc
evkmimxrt1170
mcmgr
mcmgr.c
mcmgr.h
mcmgr_internal_core_api.h
mcmgr_internal_core_api_imxrt1170.c
mcmgr_mu_internal.c
rpmsg_lite
common
llist.c
include
environment
bm
rpmsg_env_specific.h
platform
imxrt1170
rpmsg_platform.h
llist.h
rpmsg_compiler.h
rpmsg_default_config.h
rpmsg_env.h
rpmsg_lite.h
rpmsg_ns.h
virtio_ring.h
virtqueue.h
rpmsg_lite
porting
environment
rpmsg_env_bm.c
platform
imxrt1170
rpmsg_lite.c
rpmsg_ns.c
virtio
virtqueue.c
source
startup
utilities
xip
Output
erpc_matrix_multiply_rpmsg_cm7

|

Parent topic: Multicore client application

Client user code The client's user code is stored in the main_core0.c file, located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_example/erpc_matrix_multiply_rpmsg/cm7

The main_core0.c file contains the code for target board and eRPC initialization.

- After initialization, the secondary core is released from reset.
- When the secondary core is ready, the primary core initializes two matrix variables.
- The erpcMatrixMultiply eRPC function is called to issue the eRPC request and get the result.

It is possible to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in erpc_error_handler.h and erpc_error_handler.cpp files.

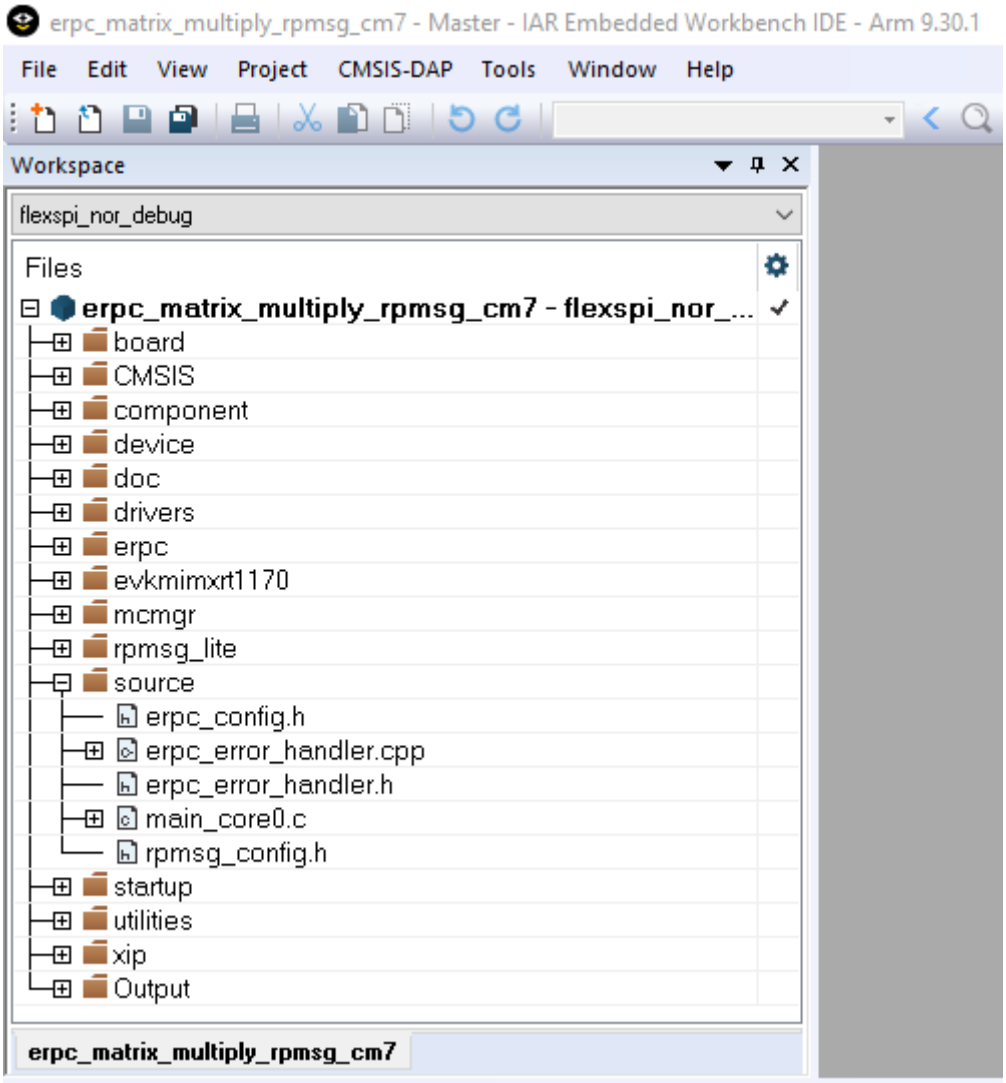
The matrix multiplication can be issued repeatedly, when pressing a software board button.

The eRPC-relevant code is captured in the following code snippet:

```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* RPMsg-Lite transport layer initialization */
erpc_transport_t transport;
transport = erpc_transport_rpmsg_lite_master_init(src, dst,
ERPC_TRANSPORT_RPMSG_LITE_LINK_ID);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_rpmsg_init(transport);
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport, message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
  /* Invoke the erpcMatrixMultiply function */
  erpcMatrixMultiply(matrix1, matrix2, result_matrix);
  ...
  /* Check if some error occured in eRPC */
  if (g_erpc_error_occurred)
  {
    /* Exit program loop */
    break;
  }
  ...
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg



|

|

Parent topic: Multicore client application

Parent topic:*Create an eRPC application*

Multiprocessor server application The "Matrix multiply" eRPC server project for multiprocessor applications is located in the *<MCUXpressoSDK_install_dir»/boards/<board_name>/multiprocessor_examples/ erpc_server_matrix_multiply_<transport_layer>* folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires server-related generated files (server shim code), server infrastructure files, and the server user code. There is no need for server multicore infrastructure files (MCMGR and RPMsg-Lite). The RPMsg-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

|SPI|<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_slave.cpp

<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.hpp

<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.cpp

| |UART|<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp

<eRPC base directory>/erpc_c/transports/erpc_uart_cmsis_transport.hpp

<eRPC base directory>/erpc_c/transports/erpc_uart_cmsis_transport.cpp

|

Server user code The server's user code is stored in the main_server.c file, located in the *<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>/* folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(Matrix matrix1, Matrix matrix2, Matrix result_matrix)
{
  ...
}
int main()
{
  ...
  /* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver␣
↪operations */
  erpc_transport_t transport;
  transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
  ...
  /* MessageBufferFactory initialization */
  erpc_mbf_t message_buffer_factory;
  message_buffer_factory = erpc_mbf_dynamic_init();
  ...
  /* eRPC server side initialization */
  erpc_server_t server;
  server = erpc_server_init(transport, message_buffer_factory);
  ...
  /* Adding the service to the server */
  erpc_service_t service = create_MatrixMultiplyService_service();
  erpc_add_service_to_server(server, service);
  ...
  while (1)
  {
    /* Process eRPC requests */
    erpc_status_t status = erpc_server_poll(server)
    /* handle error status */
    if (status != kErpcStatus_Success)
    {
      /* print error description */
      erpc_error_handler(status, 0);
      ...
    }
    ...
  }
}
```

Parent topic: Multiprocessor server application

Multiprocessor client application The "Matrix multiply" eRPC client project for multiprocessor applications is located in the *<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/iar/* folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires client-related generated files (server shim code),

client infrastructure files, and the client user code. There is no need for client multicore infrastructure files (MCMGR and RPMsg-Lite). The RPMsg-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

|SPI|<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_master.cpp

<eRPC base directory>/erpc_c/transports/ erpc_(d)spi_master_transport.hpp

<eRPC base directory>/erpc_c/transports/ erpc_(d)spi_master_transport.cpp

| |UART|<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp

<eRPC base directory>/erpc_c/transports/erpc_uart_cmsis_transport.hpp

<eRPC base directory>/erpc_c/transports/erpc_uart_cmsis_transport.cpp

|

Client user code The client's user code is stored in the main_client.c file, located in the *<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/* folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

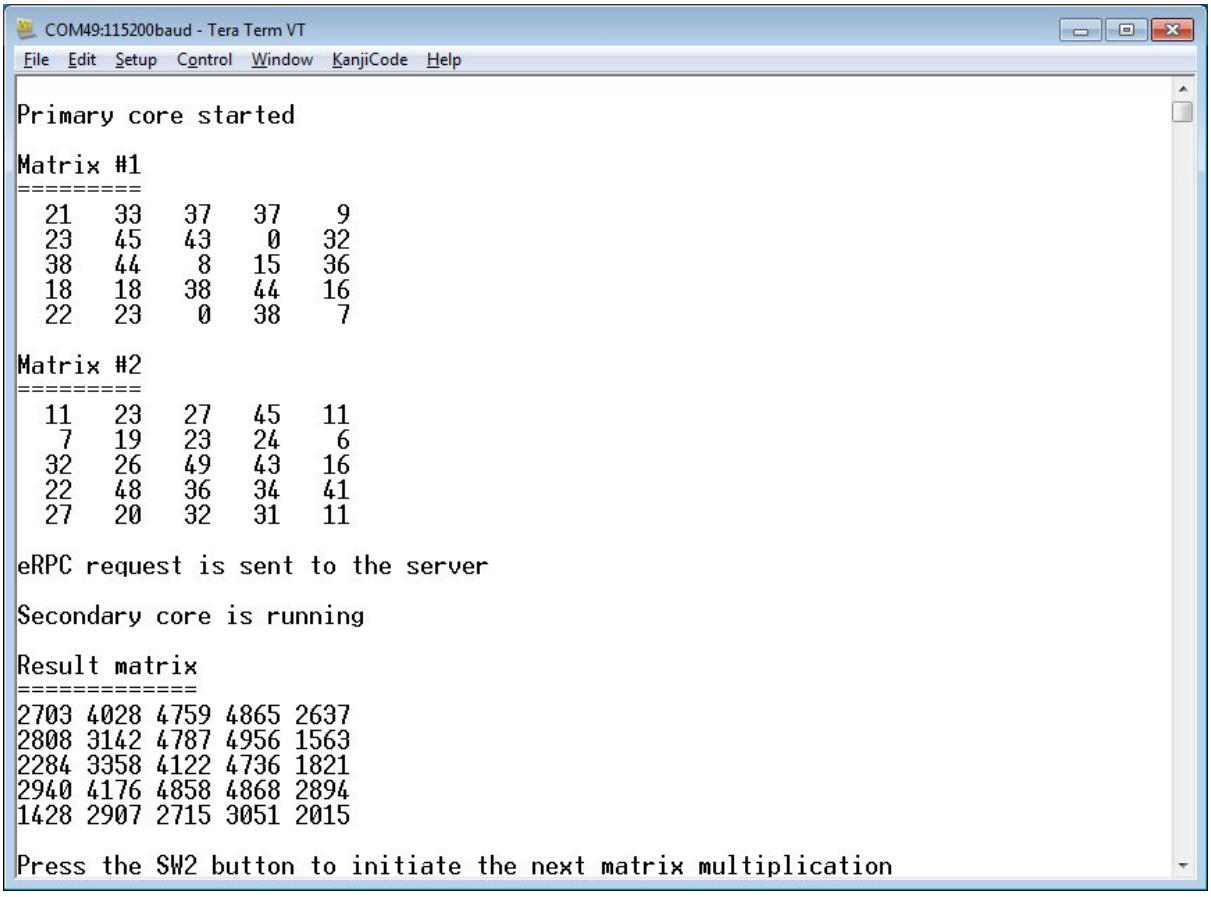
```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver␣
↪operations */
erpc_transport_t transport;
transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_dynamic_init();
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport,message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
  /* Invoke the erpcMatrixMultiply function */
  erpcMatrixMultiply(matrix1, matrix2, result_matrix);
  ...
  /* Check if some error occured in eRPC */
  if (g_erpc_error_occurred)
  {
    /* Exit program loop */
    break;
  }
  ...
}
```

Parent topic: Multiprocessor client application

Parent topic: Multiprocessor server application

Parent topic:*Create an eRPC application*

Running the eRPC application Follow the instructions in *Getting Started with MCUXpresso SDK* (document MCUXSDKGSUG) (located in the *<MCUXpressoSDK_install_dir>/docs* folder), to load both the primary and the secondary core images into the on-chip memory, and then effectively debug the dual-core application. After the application is running, the serial console should look like:



For multiprocessor applications that are running between PC and the target evaluation board or between two boards, follow the instructions in the accompanied example readme files that provide details about the proper board setup and the PC side setup (Python).

Parent topic:*Create an eRPC application*

Parent topic:*eRPC example*

Other uses for an eRPC implementation The eRPC implementation is generic, and its use is not limited to just embedded applications. When creating an eRPC application outside the embedded world, the same principles apply. For example, this manual can be used to create an eRPC application for a PC running the Linux operating system. Based on the used type of transport medium, existing transport layers can be used, or new transport layers can be implemented.

For more information and erpc updates see the github.com/EmbeddedRPC.

Note about the source code in the document Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2024 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Changelog eRPC All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

Unreleased

Added

Fixed

- Python code of the eRPC infrastructure was updated to match the proper python code style, add type annotations and improve readability.

1.14.0

Added

- Added Cmake/Kconfig support.
- Made java code jdk11 compliant, GitHub PR #432.
- Added imxrt1186 support into mu transport layer.
- erpcgen: Added assert for listType before usage, GitHub PR #406.

Fixed

- eRPC: Sources reformatted.
- erpc: Fixed typo in semaphore get (mutex -> semaphore), and write it can fail in case of timeout, GitHub PR #446.
- erpc: Free the arbitrated client token from client manager, GitHub PR #444.

- erpc: Fixed Makefile, install the erpc_simple_server header, GitHub PR #447.
- erpc_python: Fixed possible AttributeError and OSError on calling TCPTransport.close(), GitHub PR #438.
- Examples and tests consolidated.

1.13.0

Added

- erpc: Add BSD-3 license to endianness agnostic files, GitHub PR #417.
- eRPC: Add new Zephyr-related transports (zephyr_uart, zephyr_mbox).
- eRPC: Add new Zephyr-related examples.

Fixed

- eRPC,erpcgen: Fixing/improving markdown files, GitHub PR #395.
- eRPC: Fix Python client TCPTransports not being able to close, GitHub PR #390.
- eRPC,erpcgen: Align switch brackets, GitHub PR #396.
- erpc: Fix zephyr uart transport, GitHub PR #410.
- erpc: UART ZEPHYR Transport stop to work after a few transactions when using USB-CDC resolved, GitHub PR #420.

Removed

- eRPC,erpcgen: Remove cstbool library, GitHub PR #403.

1.12.0

Added

- eRPC: Add dynamic/static option for transport init, GitHub PR #361.
- eRPC,erpcgen: Winsock2 support, GitHub PR #365.
- eRPC,erpcgen: Feature/support multiple clients, GitHub PR #271.
- eRPC,erpcgen: Feature/buffer head - Framed transport header data stored in MessageBuffer, GitHub PR #378.
- eRPC,erpcgen: Add experimental Java support.

Fixed

- eRPC: Fix receive error value for spidev, GitHub PR #363.
- eRPC: UartTransport::init adaptation to changed driver.
- eRPC: Fix typo in assert, GitHub PR #371.
- eRPC,erpcgen: Move enums to enum classes, GitHub PR #379.
- eRPC: Fixed rpmsg tty transport to work with serial transport, GitHub PR #373.

1.11.0

Fixed

- eRPC: Makefiles update, GitHub PR #301.
- eRPC: Resolving warnings in Python, GitHub PR #325.
- eRPC: Python3.8 is not ready for usage of typing.Any type, GitHub PR #325.
- eRPC: Improved codec function to use reference instead of address, GitHub PR #324.
- eRPC: Fix NULL check for pending client creation, GitHub PR #341.
- eRPC: Replace sprintf with snprintf, GitHub PR #343.
- eRPC: Use MU_SendMsg blocking call in MU transport.
- eRPC: New LPSPI and LPI2C transport layers.
- eRPC: Freeing static objects, GitHub PR #353.
- eRPC: Fixed casting in deinit functions, GitHub PR #354.
- eRPC: Align LIBUSBSIO.GetNumPorts API use with libusbsio python module v. 2.1.11.
- erpcgen: Renamed temp variable to more generic one, GitHub PR #321.
- erpcgen: Add check that string read is not more than max length, GitHub PR #328.
- erpcgen: Move to g++ in pytest, GitHub PR #335.
- erpcgen: Use build=release for make, GitHub PR #334.
- erpcgen: Removed boost dependency, GitHub PR #346.
- erpcgen: Mingw support, GitHub PR #344.
- erpcgen: VS build update, GitHub PR #347.
- erpcgen: Modified name for common types macro scope, GitHub PR #337.
- erpcgen: Fixed memcpy for template, GitHub PR #352.
- eRPC,erpcgen: Change default build target to release + adding artefacts, GitHub PR #334.
- eRPC,erpcgen: Remove redundant includes, GitHub PR #338.
- eRPC,erpcgen: Many minor code improvements, GitHub PR #323.

1.10.0

Fixed

- eRPC: MU transport layer switched to blocking MU_SendMsg() API use.

1.10.0

Added

- eRPC: Add TCP_NODELAY option to python, GitHub PR #298.

Fixed

- eRPC: MUTransport adaptation to new supported SoCs.
- eRPC: Simplifying CI with installing dependencies using shell script, GitHub PR #267.
- eRPC: Using event for waiting for sock connection in TCP python server, formatting python code, C specific includes, GitHub PR #269.
- eRPC: Endianness agnostic update, GitHub PR #276.
- eRPC: Assertion added for functions which are returning status on freeing memory, GitHub PR #277.
- eRPC: Fixed closing arbitrator server in unit tests, GitHub PR #293.
- eRPC: Makefile updated to reflect the correct header names, GitHub PR #295.
- eRPC: Compare value length to used length() in reading data from message buffer, GitHub PR #297.
- eRPC: Replace EXPECT_TRUE with EXPECT_EQ in unit tests, GitHub PR #318.
- eRPC: Adapt rpmsg_lite based transports to changed rpmsg_lite_wait_for_link_up() API parameters.
- eRPC, erpcgen: Better distuingish which file can and cannot by linked by C linker, GitHub PR #266.
- eRPC, erpcgen: Stop checking if pointer is NULL before sending it to the erpc_free function, GitHub PR #275.
- eRPC, erpcgen: Changed api to count with more interfaces, GitHub PR #304.
- erpcgen: Check before reading from heap the buffer boundaries, GitHub PR #287.
- erpcgen: Several fixes for tests and CI, GitHub PR #289.
- erpcgen: Refactoring erpcgen code, GitHub PR #302.
- erpcgen: Fixed assigning const value to enum, GitHub PR #309.
- erpcgen: Enable runTesttest_enumErrorCode_allDirection, serialize enums as int32 instead of uint32.

1.9.1

Fixed

- eRPC: Construct the USB CDC transport, rather than a client, GitHub PR #220.
- eRPC: Fix premature import of package, causing failure when attempting installation of Python library in a clean environment, GitHub PR #38, #226.
- eRPC: Improve python detection in make, GitHub PR #225.
- eRPC: Fix several warnings with deprecated call in pytest, GitHub PR #227.
- eRPC: Fix freeing union members when only default need be freed, GitHub PR #228.
- eRPC: Fix making test under Linux, GitHub PR #229.
- eRPC: Assert costumizing, GitHub PR #148.
- eRPC: Fix corrupt clientList bug in TransportArbitrator, GitHub PR #199.
- eRPC: Fix build issue when invoking g++ with -Wno-error=free-nonheap-object, GitHub PR #233.
- eRPC: Fix inout cases, GitHub PR #237.

- eRPC: Remove ERPC_PRE_POST_ACTION dependency on return type, GitHub PR #238.
- eRPC: Adding NULL to ptr when codec function failed, fixing memcpy when fail is present during deserialization, GitHub PR #253.
- eRPC: MessageBuffer usage improvement, GitHub PR #258.
- eRPC: Get rid for serial and enum34 dependency (enum34 is in python3 since 3.4 (from 2014)), GitHub PR #247.
- eRPC: Several MISRA violations addressed.
- eRPC: Fix timeout for Freertos semaphore, GitHub PR #251.
- eRPC: Use of rpmsg_lite_wait_for_link_up() in rpmsg_lite based transports, GitHub PR #223.
- eRPC: Fix codec nullptr dereferencing, GitHub PR #264.
- erpcgen: Fix two syntax errors in erpcgen Python output related to non-encapsulated unions, improved test for union, GitHub PR #206, #224.
- erpcgen: Fix serialization of list/binary types, GitHub PR #240.
- erpcgen: Fix empty list parsing, GitHub PR #72.
- erpcgen: Fix templates for malloc errors, GitHub PR #110.
- erpcgen: Get rid of encapsulated union declarations in global scale, improve enum usage in unions, GitHub PR #249, #250.
- erpcgen: Fix compile error:UniqueIdChecker.cpp:156:104:'sort' was not declared, GitHub PR #265.

1.9.0

Added

- eRPC: Allow used LIBUSBSIO device index being specified from the Python command line argument.

Fixed

- eRPC: Improving template usage, GitHub PR #153.
- eRPC: run_clang_format.py cleanup, GitHub PR #177.
- eRPC: Build TCP transport setup code into liberpc, GitHub PR #179.
- eRPC: Fix multiple definitions of g_client error, GitHub PR #180.
- eRPC: Fix memset past end of buffer in erpc_setup_mbf_static.cpp, GitHub PR #184.
- eRPC: Fix deprecated error with newer pytest version, GitHub PR #203.
- eRPC, erpcgen: Static allocation support and usage of rpmsg static FreeRTOSs related APi, GitHub PR #168, #169.
- erpcgen: Remove redundant module imports in erpcgen, GitHub PR #196.

1.8.1

Added

- eRPC: New i2c_slave_transport trasnport introduced.

Fixed

- eRPC: Fix misra erpc c, GitHub PR #158.
- eRPC: Allow conditional compilation of message_loggers and pre_post_action.
- eRPC: (D)SPI slave transports updated to avoid busy loops in rtos environments.
- erpcgen: Re-implement EnumMember::hasValue(), GitHub PR #159.
- erpcgen: Fixing several misra issues in shim code, erpcgen and unit tests updated, GitHub PR #156.
- erpcgen: Fix bison file, GitHub PR #156.

1.8.0

Added

- eRPC: Support win32 thread, GitHub PR #108.
- eRPC: Add mbed support for malloc() and free(), GitHub PR #92.
- eRPC: Introduced pre and post callbacks for eRPC call, GitHub PR #131.
- eRPC: Introduced new USB CDC transport.
- eRPC: Introduced new Linux spidev-based transport.
- eRPC: Added formatting extension for VSC, GitHub PR #134.
- erpcgen: Introduce ustring type for unsigned char and force cast to char*, GitHub PR #125.

Fixed

- eRPC: Update makefile.
- eRPC: Fixed warnings and error with using MessageLoggers, GitHub PR #127.
- eRPC: Extend error msg for python server service handle function, GitHub PR #132.
- eRPC: Update CMSIS UART transport layer to avoid busy loops in rtos environments, introduce semaphores.
- eRPC: SPI transport update to allow usage without handshaking GPIO.
- eRPC: Native _WIN32 erpc serial transport and threading.
- eRPC: Arbitrator deadlock fix, TCP transport updated, TCP setup functions introduced, GitHub PR #121.
- eRPC: Update of matrix_multiply.py example: Add –serial and –baud argument, GitHub PR #137.
- eRPC: Update of .clang-format, GitHub PR #140.
- eRPC: Update of erpc_framed_transport.cpp: return error if received message has zero length, GitHub PR #141.
- eRPC, erpcgen: Fixed error messages produced by -Wall -Wextra -Wshadow -pedantic-errors compiler flags, GitHub PR #136, #139.
- eRPC, erpcgen: Core re-formatted using Clang version 10.
- erpcgen: Enable deallocation in server shim code when callback/function pointer used as out parameter in IDL.
- erpcgen: Removed ‘$’ character from generated symbol name in ‘_$union’ suffix, GitHub PR #103.

- erpcgen: Resolved mismatch between C++ and Python for callback index type, GitHub PR #111.
- erpcgen: Python generator improvements, GitHub PR #100, #118.
- erpcgen: Fixed error messages produced by -Wall -Wextra -Wshadow -pedantic-errors compiler flags, GitHub PR #136.

1.7.4

Added

- eRPC: Support MU transport unit testing.
- eRPC: Adding mbed os support.

Fixed

- eRPC: Unit test code updated to handle service add and remove operations.
- eRPC: Several MISRA issues in rpmsg-based transports addressed.
- eRPC: Fixed Linux/TCP acceptance tests in release target.
- eRPC: Minor documentation updates, code formatting.
- erpcgen: Whitespace removed from C common header template.

1.7.3

Fixed

- eRPC: Improved the test_callbacks logic to be more understandable and to allow requested callback execution on the server side.
- eRPC: TransportArbitrator::prepareClientReceive modified to avoid incorrect return value type.
- eRPC: The ClientManager and the ArbitratedClientManager updated to avoid performing client requests when the previous serialization phase fails.
- erpcgen: Generate the shim code for destroy of statically allocated services.

1.7.2

Added

- eRPC: Add missing doxygen comments for transports.

Fixed

- eRPC: Improved support of const types.
- eRPC: Fixed Mac build.
- eRPC: Fixed serializing python list.
- eRPC: Documentation update.

1.7.1

Fixed

- eRPC: Fixed semaphore in static message buffer factory.
- erpcgen: Fixed MU received error flag.
- erpcgen: Fixed tcp transport.

1.7.0

Added

- eRPC: List names are based on their types. Names are more deterministic.
- eRPC: Service objects are as a default created as global static objects.
- eRPC: Added missing doxygen comments.
- eRPC: Added support for 64bit numbers.
- eRPC: Added support of program language specific annotations.

Fixed

- eRPC: Improved code size of generated code.
- eRPC: Generating crc value is optional.
- eRPC: Fixed CMSIS Uart driver. Removed dependency on KSDK.
- eRPC: Forbid users use reserved words.
- eRPC: Removed outByref for function parameters.
- eRPC: Optimized code style of callback functions.

1.6.0

Added

- eRPC: Added @nullable support for scalar types.

Fixed

- eRPC: Improved code size of generated code.
- eRPC: Improved eRPC nested calls.
- eRPC: Improved eRPC list length variable serialization.

1.5.0

Added

- eRPC: Added support for unions type non-wrapped by structure.
- eRPC: Added callbacks support.
- eRPC: Added support @external annotation for functions.
- eRPC: Added support @name annotation.
- eRPC: Added Messaging Unit transport layer.
- eRPC: Added RPMSG Lite RTOS TTY transport layer.
- eRPC: Added version verification and IDL version verification between eRPC code and eRPC generated shim code.
- eRPC: Added support of shared memory pointer.
- eRPC: Added annotation to forbid generating const keyword for function parameters.
- eRPC: Added python matrix multiply example.
- eRPC: Added nested call support.
- eRPC: Added struct member "byref" option support.
- eRPC: Added support of forward declarations of structures
- eRPC: Added Python RPMsg Multiendpoint kernel module support
- eRPC: Added eRPC sniffer tool

1.4.0

Added

- eRPC: New RPMsg-Lite Zero Copy (RPMsgZC) transport layer.

Fixed

- eRPC: win_flex_bison.zip for windows updated.
- eRPC: Use one codec (instead of inCodec outCodec).

[1.3.0]

Added

- eRPC: New annotation types introduced (@length, @max_length, ...).
- eRPC: Support for running both erpc client and erpc server on one side.
- eRPC: New transport layers for (LP)UART, (D)SPI.
- eRPC: Error handling support.

[1.2.0]

Added

- eRPC source directory organization changed.
- Many eRPC improvements.

[1.1.0]

Added

- Multicore SDK 1.1.0 ported to KSDK 2.0.0.

[1.0.0]

Added

- Initial Release

3.3 Wireless

3.3.1 NXP Wireless Framework and Stacks

Wireless Zigbee

MCXW71/MCXW72/RW612 ZigBee Demo Applications User Guide

Overview This document applies to the MCXW71/MCXW72/RW612 device ZigBee 3.x wireless microcontrollers, referred to as ZigBee devices throughout this document.

This document provides details of the ZigBee example applications. These examples demonstrate the features and operation of the base device in a ZigBee 3.x network that uses NXP microcontrollers from the MCXW71/MCXW72/RW612 families. The examples can be a starting point for developing real-world devices.

Related documentation The table below lists the documents that can be referred for more information and developing custom applications based on this user guide.

Related documentation

Documents	Description	Link/how to obtain
ZigBee Base Device Behavior Specification Version 1.0	It provides a definition for the base device behavior specification, for devices operating on the ZigBee-PRO stack.	ZigBee Base Device Behavior Specification
Getting Started with the FRDM-MCXW71	It provides detailed instructions for installing the MCUXpresso SDK for the FRDM-MCXW71 board.	Getting Started with FRDM-MCXW71
Getting Started with the FRDM-MCXW72	It provides detailed instructions for installing the MCUXpresso SDK for the FRDM-MCXW72 board.	Getting Started with the FRDM-MCXW72 Development Platform
Getting Started with the MCX-W71-EVK	It provides detailed instructions for installing the MCUXpresso SDK for the MCX-W71-EVK board.	Getting Started with the MCX-W71 Development Platform
Getting Started with the MCX-W72-EVK	It provides detailed instructions for installing the MCUXpresso SDK for the MCX-W72-EVK board.	Getting Started with the MCX-W72 Development Platform
Getting Started with the FRDM-RW612 board	It provides detailed instructions for installing the MCUXpresso SDK for the FRDM-RW612 board.	Getting Started with the FRDM-RW612 Development Platform
ZigBee 3.0 Stack User Guide (JNUG 3130)	It provides information relating to the ZigBee 3.0 wireless networking protocol and its associated stack for implementation on NXP microcontrollers.	Contact an NXP field applications engineer (FAE) or sales representative
ZigBee 3.0 Stack User Guide (JNUG 3130)	It provides information relating to the ZigBee 3.0 wireless networking protocol and its associated stack for implementation on NXP microcontrollers.	Contact an NXP field applications engineer (FAE) or sales representative
ZigBee 3.0 Devices User Guide (JNUG3131)	It introduces and provides details of the ZigBee Base Devices.	Contact an NXP field applications engineer (FAE) or sales representative
ZigBee 3.0 Cluster Library User Guide (JNUG3132)	It describes the NXP implementation of the ZigBee Cluster Library (ZCL) for the ZigBee 3.0 standard.	Contact an NXP field applications engineer (FAE) or sales representative
Core Utilities User Guide [JNUG3133]	It describes the device Core Utilities (JCU) that is used in wireless network applications for the NXP device-based microcontrollers.	Contact an NXP field applications engineer (FAE) or sales representative
ZigBee 3.0 Green Power User Guide [JNUG3134]	It describes the use of the NXP implementation of the Green Power feature for ZigBee 3.0 applications.	Contact an NXP field applications engineer (FAE) or sales representative

Parent topic:*Overview*

Introduction A ZigBee 3.x wireless network comprises various ZigBee software devices that are implemented on hardware platforms to form nodes. These ZigBee examples are concerned with implementing the ZigBee Base device on the NXP ZigBee device (MCXW71/MCXW72/RW612).

This document provides example implementations of the following ZigBee logical device types:

- Coordinator
- Router

- End device with Receiver always On
- End device with Receiver Off

The examples of the above device types are not real-world devices but provide the basic behavior required by the ZigBee Base Device Behavior Specification. These examples must serve as base templates for further development into real physical devices. The ZigBee Base Device is introduced and detailed in the *ZigBee 3.0 Devices User Guide* (document JNUG3131).

The ZigBee Base Device Behavior Specification provides definitions, procedures and methods for forming, joining, and maintaining ZigBee 3.x networks. It also defines the method for service discovery, which binds a client and server of an operational cluster to achieve the functionality of the physical devices. For more information on ZigBee 3.x networks general introduction, refer to the *ZigBee 3.0 Stack User Guide* (document JNUG3130).

Development environment This section includes software and hardware requirements:

- Software
- Hardware

Software To use the ZigBee examples, install the following software:

- MCUXpresso Integrated Design Environment (IDE)
- MCXW71/MCXW72 ZigBee 3.0 Software Development Kit (SDK)
- Python 3
- Required python modules: lxml, openpyxl, SPSDK version 1.11.0 and crccheck
- Install all required Python modules from ${SDK_ROOT}/middleware/wireles/zigbee:
  `$ pip install -r requirements.txt`

  Note: Both SPSDK version 1.11.0 and crccheck modules are required to generate OTA images using the NXP ZB OTA tool (`nxpzbota.py`).

The MCUXpresso software and installation instructions are described in Getting Started with the FRDM-MCXW71 Development Platform.

Support for Zigbee packet sniffing is provided by using the `Sniffer_1000000baud_8N1_NoFlowControl` sniffer binary available in the `tools/sniffer` K32W061 SDK. The binary must be flashed on a K32W0 board using either of the following two methods:

- Using the *Getting Started with MCUXpresso SDK for K32W061* (document MCUXSDKK32W061GSUG). For more information, see the section "Building and Flashing the Application". In this case, the user must also install the K32W061 SDK.
- Using the firmware loader from NXP Test Tool.

Also, ensure to install the following tools:

- Kinetis Protocol Analyzer Adapter 2.0.3.1 or newer and Wireshark
- J-Link software, which can be downloaded from J-Link / J-Trace Downloads

The wireless microcontroller-specific resources and documentation are available via the MCUXpresso to authorized users.

Parent topic:*Development environment*

Hardware NXP enables the development of ZigBee 3.x applications by providing supported hardware kits. The following boards provide a platform for running the Zigbee applications:

- FRDM-MCXW71 board

- FRDM-MCXW72 board
- MCX-W71-EVK board
- MCX-W72-EVK board
- FRDM-RW612 board
- RD-RW612-BGA board

Parent topic:*Development environment*

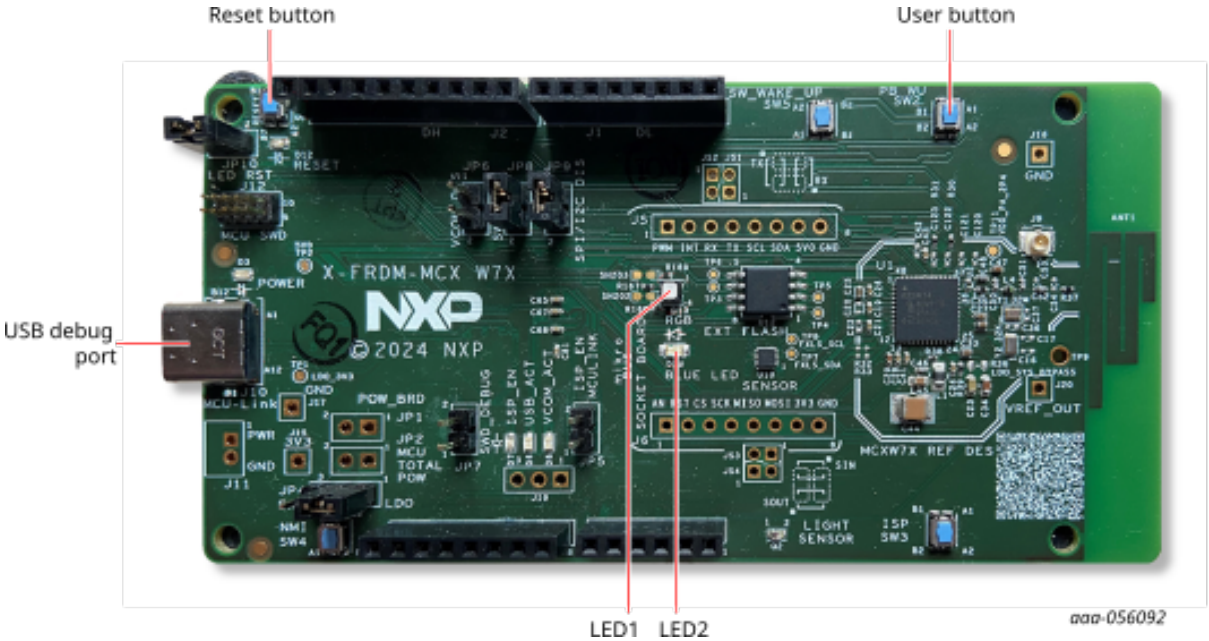
Examples The table below lists the example applications provided.

Application	Device type
Coordinator	Coordinator/Trust center
Router	Router
End Device RX On	End Device (Non-sleeping)
End Device RX Off when Idle	End Device (Sleeping)

The examples are provided as part of the wireless examples for ZigBee. The binaries generated by these examples are targeted to run on the FRDM-MCXW71 board, FRDM-MCXW72, MCX-W71-EVK, MCX-W72-EVK, RD-RW612-BGA, and FRDM-RW612 boards.

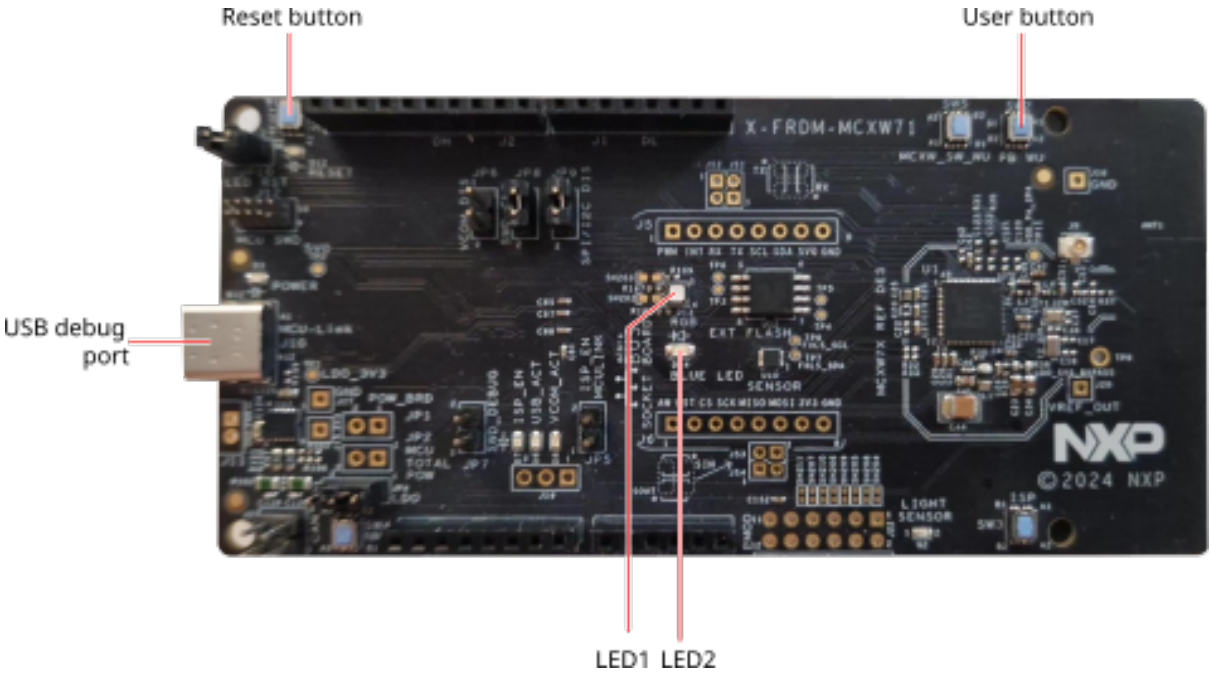
Note: *The RW612 platform does not support the RX OFF Application use case currently.*

FRDM-MCXW7x board The FRDM-MCXW7x board is used for the ZigBee examples described in this document. The important LEDs and user control buttons are highlighted, as shown in the figure, **FRDM-MCXW7x board** below.



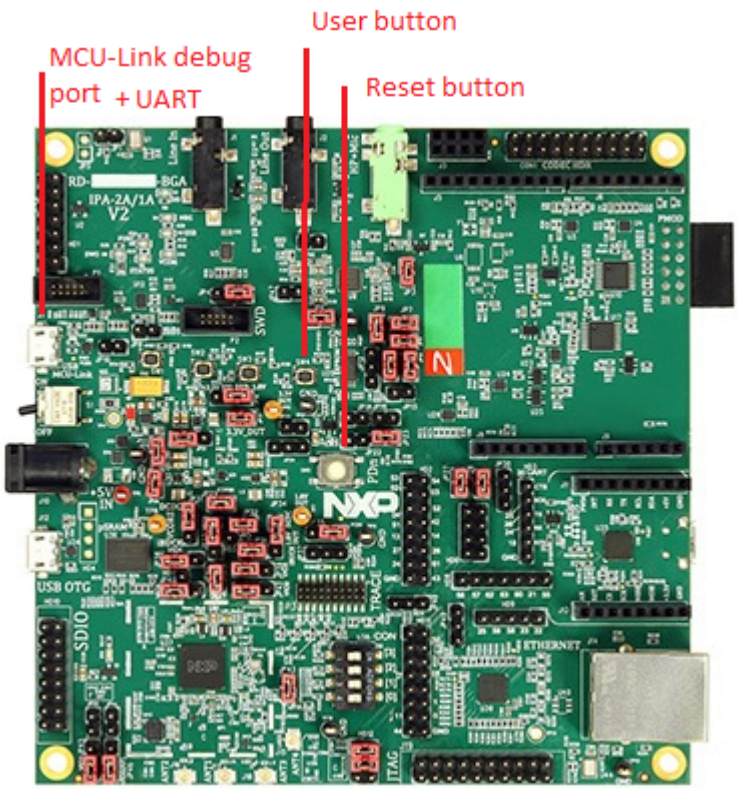
Parent topic:*Examples*

FRDM-MCXW71 / FRDM-MCXW72 board The FRDM-MCXW71 or the FRDM-MCXW72 board can be used for the ZigBee examples described in this document. The important LEDs and user control buttons are highlighted, as shown in the figure below.



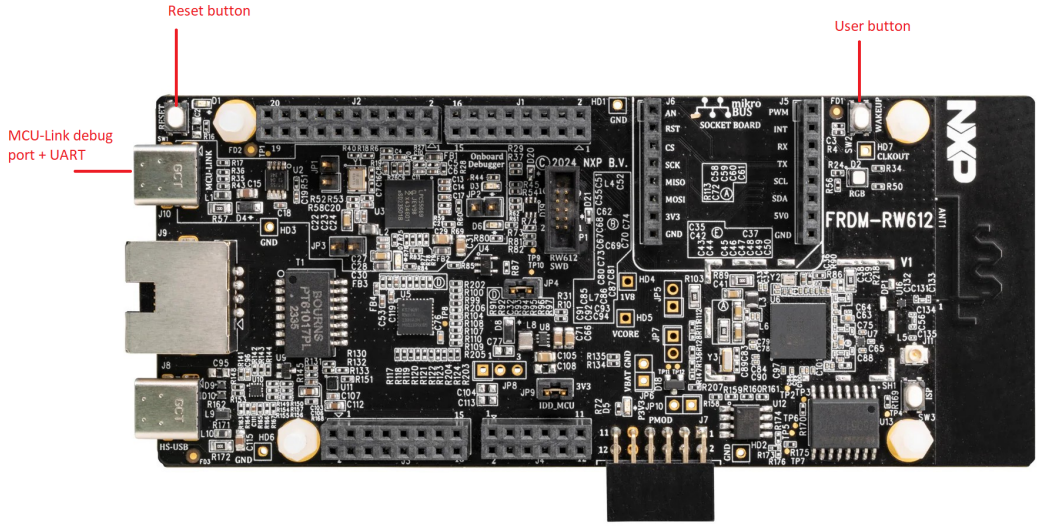
Parent topic:*Examples*

RD-RW612-BGA board The FRDM-RW612 board can be used for the ZigBee examples described in this document. The important LEDs and user control buttons are highlighted, as shown in the figure, **RD-RW612-BGA board** below.



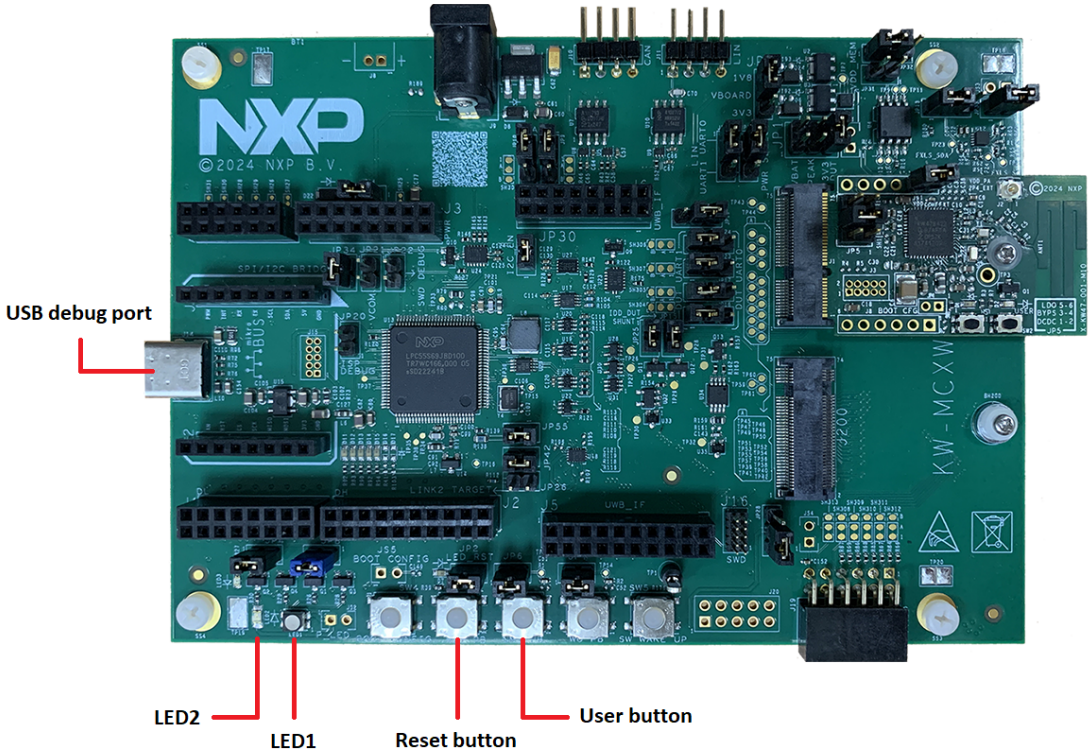
Parent topic:*Examples*

FRDM-RW612 board The FRDM-RW612 board is used for the ZigBee examples described in this document. The important LEDs and user control buttons are highlighted, as shown in the figure, **FRDM-RW612 board** below.



Parent topic:*Examples*

MCX-W71-EVK / MCX-W72-EVK board The MCX-W71-EVK or the MCX-W72-EVK board can be used for the ZigBee examples described in this document. The board is shown in the figure below.



Parent topic:*Examples*

Running the demonstration examples This section describes how to use the supplied pre-built binaries to run the example applications on a ZigBee device. All the applications run on the FRDM-MCXW71 board (or the FRDM-MCXW72 / MCX-W71-EVK / MCX-W72-EVK / FRDM-RW612 / RD-RW612-BGA boards). The examples do not run on other development kits. All the ZigBee wireless demo applications are configured to use a default channel 12. The pre-processor configuration uses the setting SINGLE_CHANNEL=12, which can be overwritten to change it to a different channel.

The ZigBee examples work in the Centralized (Trust center) network only, and all devices are expected to join with the ZigBee09 key. Once joined, the devices then automatically negotiate a new TCLK key.

For details of the differences between a Centralized (Trust center) network and a Distributed network, see the *ZigBee Devices User Guide* (document JNUG3131).

Loading the applications The table below lists the binaries generated by the examples.

Hardware platform	Binary files
FRDM-MCXW board	• frdmmcxw71_zigbee_coordinator_bm.axf• frdmmcxw71_zigbee_router_bm.axf• frdmmcxw71_zigbee_ed_rx_on_bm.axf• frdmmcxw71_zigbee_ed_rx_off_bm.axf• frdmmcxw71_zigbee_coordinator_freertos.axf• frdmmcxw71_zigbee_router_freertos.axf• frdmmcxw71_zigbee_ed_rx_on_freertos.axf• frdmmcxw71_zigbee_ed_rx_off_freertos.axf
FRDM-MCXW board	• frdmmcxw72_zigbee_coordinator_freertos.axf• frdmmcxw72_zigbee_router_freertos.axf• frdmmcxw72_zigbee_ed_rx_on_freertos.axf• frdmmcxw72_zigbee_ed_rx_off_freertos.axf
MCX-W71-EVK board	• mcxw71evk_zigbee_coordinator_bm.axf• mcxw71evk_zigbee_router_bm.axf• mcxw71evk_zigbee_e
MCX-W72-EVK board	• mcxw72evk_zigbee_coordinator_freertos.axf• mcxw72evk_zigbee_router_freertos.axf• mcxw72evk_

To write the images to the board, perform the following steps:

1. Use the J-Link utility.
2. Download the J-Link from J-Link / J-Trace Downloads.
3. Plug the FRDM-MCXW71 board (or FRDM-MCXW72, or MCX-W71-EVK board, or MCX-W72-EVK board) to the USB port (do not keep the **SW4** button pressed while doing this step).
4. Create a commands_script file with the following content (change the application name as necessary):

```
Reset
Halt
Erase
LoadFile <PATH_TO_AXF>
Reset
Go
Quit
```

Note: Make sure to always perform **Erase** step when switching between **R22** and **R23** applications.

5. Copy the application and commands_script in the same folder where the J-Link executable is placed.
6. Run the following code:

```
JLink.exe/JLinkExe (linux) -<DEVICE_NAME> -if SWD -speed 4000 -autoconnect 1 -
↪CommanderScript commands_script
```

   where DEVICE_NAME can be:

   - MCXW716 for MCXW71
   - MCXW727C_CORE0 for MCXW72

After completing the above steps, reset the board or module to run the application.

Loading the application for RW612 platform For instructions on loading the examples for the RD-RW612-BGA and FRDM-RW612 boards, refer to the section RW612: loading the applications.

Refer to the Over The Air Upgrade (OTA) for RW612 platforms.

Note: The RW612 platform only supports the west build.

Parent topic:*Running the demonstration examples*

Coordinator functionality The functionality of the Coordinator application is described as follows:

- The Coordinator is responsible for initially forming the network. It manages other devices that can join the network via the Trust center functionality. It distributes security materials to those devices that are allowed to join. The Coordinator supports the mandatory clusters and features of the Base Device as defined in the *ZigBee Base Device Behavior Specification*.
- The Coordinator also supports the On/Off Cluster as a client for demonstrating the "Finding and Binding" functionality.
- The serial commands issued from a terminal program control the Coordinator. The terminal program runs on a PC connected to the Zigbee device through a USB connection. The Coordinator application is configured to communicate with the following configuration:
  - Baud rate = 115200
  - Data = 8 bits
  - Stop = 1 bit
  - Parity = None
  - Flow control = None The serial interface is not case-sensitive. For a summary of the serial interface, refer to Summary of Serial Interface Commands.

Forming a network A network can be formed from a factory-new Coordinator (Network Steering while not on a network) as follows:

1. Enter the form on the serial interface. The Coordinator then starts a network.
2. Using a ZigBee packet sniffer (running separately on a USB Dongle), validates the user regarding network status. The periodic "link status" messages must be present on the operational channel. This step is optional.

Parent topic: Coordinator functionality

Allowing other nodes to join (Network Steering) Once a network has been formed, it must be opened to allow other devices to join it, referred to as Network Steering while on a network. To initiate Network Steering, perform the following steps:

1. Enter the steer on the serial interface (Dongle or Carrier Board).
2. Then, the Coordinator broadcasts a Management Permit Join Request to the network to open the "permit join" window for 180 seconds.
3. The Network Steering process (for devices not on a network) can now be triggered on the devices that are to join the network.

Parent topic: Coordinator functionality

Operating the device (Coordinator) The operational functionality of this device in this demonstration is provided by the On/Off cluster. Enter the toggle in the serial interface (Carrier Board) to send an OnOff Toggle command to the bound device (in the Binding table).

Parent topic: Coordinator functionality

Rejoining a network (Coordinator) As a Coordinator, when this device is restarted in a state that is not factory-new, it resumes operation in its previous state. All applications, bindings, groups, and network parameters are preserved in non-volatile memory.

Parent topic: Coordinator functionality

Performing a factory reset (Coordinator) The Coordinator can be returned to its factory-new state, which erases all persistent data except the outgoing network frame counter.

To perform a factory reset, enter the following command on the serial interface:

```
factory reset
```

Parent topic: Coordinator functionality

Summary of serial interface commands The serial port connection to the Coordinator application is set up with the following configuration settings:

- Baud rate = 115200
- Data = 8 bits
- Stop = 1 bit
- Parity = None

The serial commands are not case-sensitive.

Serial command	Action
toggle	Sends an On/Off Toggle command to bound devices
steer	Triggers Network Steering for a device on the network
form	Triggers network formation for a device not on a network
find	Triggers Finding and Binding as an initiator
factory reset	Factory resets the device, erasing persistent data
soft reset	Triggers a software reset (no loss of data)

Parent topic: Coordinator functionality

Parent topic:*Running the demonstration examples*

Router functionality For demonstrating the "Finding and Binding" functionality, the Router also supports the On/Off Cluster as a server.

Forming or joining a network (router) The router can only join an existing network. If it does not find the network, it continues discovering the network until it can find a network to join.

Parent topic: Router functionality

Joining an existing network using network steering (Router) A factory-new Router can join an existing ZigBee only when the network is opened to accept new joiners (Network Steering for a device on a network).

Joining an existing network using Network Steering is achieved as follows:

1. Trigger Network Steering on one of the devices already on the network (Coordinator or another Router in the same ZigBee network).
2. Then reset using the **RESET** button or power on the joining Router device.
3. As a result, the Router starts a network discovery and the associate process. Association is followed by an exchange of security materials and an update of the Trust center link key (if joining a Centralized Trust center network).
4. By power cycling, the join can be retried if it fails.

Parent topic: Router functionality

Allowing other devices to join the network (Router) Once the Router joins a network, the network is immediately opened for 180 seconds to allow other devices to join (Network Steering while on a network). During this time, new devices can join the network using either the Coordinator or the Router as the parent, with no additional steps required. After this time expires, no new nodes can ever join the network. If rejoining other nodes is necessary after this time expires, perform the following steps:

1. Press the **USER** button on the (FRDM-MCXW71 / FRDM-MCXW72 / MCX-W71-EVK / MCX-W72-EVK / FRDM-RW612 / RD-RW612-BGA board). The same button is also used to start "*Finding and Binding*", described in Binding devices.
2. Then, the Router broadcasts a Management Permit Join Request to the network to open the "permit join" window for 180 seconds. The Network Steering process (for devices not on a network) can now be triggered on the devices that are to join the network.

Parent topic: Router functionality

Operating the device (router) The operational functionality of this device in this demonstration is provided by the On/Off cluster. As the device supports the On/Off cluster server, its operation is passive, and it responds to commands sent by bound devices. It responds to an OnOff Toggle command from a bound controller device, by toggling the LED1 on the FRDM-MCXW71, FRDM-MCXW72, MCX-W72-EVK, MCX-W72-EVK, RD-RW612-BGA, or the FRDM-RW612 board.

Parent topic: Router functionality

Rejoining a network (router) As a Router, when this device is restarted in a state, which is not factory-new, it resumes operation in its previous state. All applications, binding, group, and network parameters are preserved within the non-volatile memory of the device.

Parent topic: Router functionality

Performing a factory reset (router) The Router can be returned to its factory-new state (erasing all persistent data except the outgoing network frame counter) as follows:

- Hold down the **USER** button and press the **RESET** button on the board (FRDM-MCXW71 / FRDM-MCXW72 / MCX-W71-EVK / MCX-W72-EVK / FRDM-RW612 / RD-RW612-BGA).

The Router then broadcasts a Leave Indication on the old network. It also deletes all persistent data (except the outgoing network frame counter) and performs a software reset.

The two supported over-the-air commands for removing a device from the network are as follows:

- Network Leave Request without rejoin
- ZDO Management Network Leave Request without rejoin

The Reset command of the Basic cluster causes the ZCL to be reset to its factory-new defaults, resetting all attributes and configured reports. It does not remove the device from the network. Therefore, all network parameters, groups, and bindings remain in place.

Parent topic: Router functionality

Parent topic:*Running the demonstration examples*

End Device functionality The End Device is not capable of either forming a network or being a parent to other devices joining the network.

Two types of End Device are as follows:

- "RX On" End Devices, which are always ready to communicate in the network
- Sleepy "RX Off when Idle" End Devices, which can sleep for periods of time during which it cannot communicate

Note: To disable sleep on "RX Off" End Device for continuous data polling, user must define **DISABLE_SLEEP** in the corresponding ed_rx_off.cmake (by default, it is undefined).

The End Device supports the mandatory clusters and features of the Base Device as defined in ZigBee Base Device Behavior Specification.

For demonstrating the "Finding and Binding" functionality, the End Device also supports the On/Off cluster as a client.

All communications to/from the End Device are passed through its parent Coordinator or Router. For an RX Off with the Idle End Device, communication is initiated from the End Device through Poll Requests. The parent device then buffers data for the child End Device for some time. During this period, the End Device must send periodic Poll Requests to its parent to receive any messages that are waiting for it.

For the RX On device, no poll is required and messages are sent directly from the parent. However, regular messages must be sent. Otherwise, the device can be timed out.

Joining an existing network using network steering (ED) A factory-new End Device can join an existing network once the network opens to accept new joiners (Network Steering for a device on a network).

Joining an existing network using Network Steering is achieved as follows:

1. Trigger Network Steering on one of the devices already on the network.
2. Then, reset using the **RESET** button or power on the End device.

Parent topic: End Device functionality

Operating the device (ED) The operational functionality of this device in this demonstration is provided by the On/Off cluster. The device supports the On/Off cluster server. Therefore, its operation is passive and it responds to commands sent by bound devices. It responds to an OnOff Toggle command from a bound controller device, by toggling the **LED1** on the FRDM-MCXW71, FRDM-MCXW72, MCX-W71-EVK, MCX-W72-EVK, RD-RW612-BGA, or the FRDM-RW612 board.

Parent topic: End Device functionality

Rejoining a network (ED) An End Device can be restarted in a state, which is not factory-new. In such a case, it automatically sends a Network Rejoin Request to re-establish contact with its previous parent. If this fails, it then tries to join any Router on the network that can host it. The End Device attempts to rejoin when powered on and after it wakes from the deep sleep state. All the application, binding, group, and network parameters are preserved in non-volatile memory.

Parent topic: End Device functionality

Performing a factory reset (ED) The End Device can be returned to its factory-new state (erasing all persistent data except the outgoing network frame counter) as follows:

- Hold down the **USER** button and press the **RESET** button on the FRDM-MCXW71 board or FRDM-MCXW72 board or MCX-W71-EVK board, or MCX-W72-EVK board.

The End Device then unicasts a Leave Indication to its parent. The parent then broadcasts this message again to the old network. The End Device deletes all persistent data (other than the outgoing network frame counter) and performs a software reset.

There are two supported over-the-air commands for removing a device from the network as follows:

- Network Leave Request without rejoin
- ZDO Management Network Leave Request without rejoin

The Reset command of the Basic cluster causes the ZCL to be reset to its factory-state defaults. It also resets all attributes and configured reports. This step does not remove the device from the network and all network parameters, groups, and bindings remain in place.

Parent topic: End Device functionality

Parent topic:*Running the demonstration examples*

Binding devices The Router and End Device support the On/Off cluster as a server and implement the "Finding and Binding" process as a target.

To trigger "Finding and Binding" as a target, perform the following steps:

1. Press the **USER** button on the FRDM-MCXW71 board (or the FRDM-MCXW72 / MCX-W71-EVK / MCX-W72-EVK / FRDM-RW612 / RD-RW612-BGA board) of the target device. The same button is used to start Network Steering, described in Allowing Other Devices to Join the Network.
2. Start "*Finding and Binding*" on the initiator device.

This step causes the End Device or Router to self-identify for 180 seconds. In this duration, the initiator tries to find the identifying devices, queries their capabilities, and creates bindings on the devices with matching operational clusters. As part of this process, the Route or End Device can receive an Add Group command and/or a Binding Request command.

Reporting is a mandatory feature in ZigBee 3.x. The Router and End Device supports the On/Off cluster as a server and the OnOff attribute of this cluster is a reportable attribute as defined in ZigBee Base Device Behavior Specification. The Router and End Device hold a default configuration for reporting the state of the OnOff attribute. Once a device wishing to receive these periodic and on-change reports creates a remote binding, the Router starts to send reports to

this bound device. The frequency of the reports depends on the default report configuration of the individual target device; 60 seconds in this case. The device receiving the reports can request the change by sending a Report Configuration command.

Parent topic:*Running the demonstration examples*

ZigBee over-the-air upgrade An over-the-air (OTA) upgrade involves transferring a new firmware image to a device already installed and operational within a ZigBee network. This functionality is provided by the OTA upgrade cluster. To upgrade the devices on a network, two functional elements are required as follows:

- **OTA Server:** First, the network must host an OTA server, which receives new OTA images from manufacturers and advertise the OTA image details to the network. Then, it must deliver the new image to the requested devices.
- **OTA Clients:** The second requirement is for OTA clients, which are on the network devices that can be updated. These devices periodically interrogate the OTA server for details of the firmware images available. If a client finds a suitable upgrade image on the server, it starts to request this image, storing each part as it is received. Once the full image has been received, it validates, and the device boots to run the new image.

The clients pull down the new images, requesting each block in turn and filling in the gaps. The server never pushes the images onto the network.

OTA for RW612 platform For information about the over-the-air upgrade process for RW612 platforms, refer to the section Zigbee Over The Air Upgrade (OTA) for RW612 platforms.

Overview Support for the OTA upgrade cluster as a client has been included for the Router and End Device.

Note: By default, all the devices only support encrypted OTA.

The internal flash memory is used to store the upgraded image by default.

The table below shows that the initial client binaries to be programmed into the MCXW71, and MCXW72 devices are version 1 files.

Version 1 files

Hardware platform	Binary files
FRDM-MCXW71 board	• frdmmcxw71_zigbee_router_bm_v1.axf• frdmmcxw71_zigbee_ed_rx_on_bm_v1.axf• frdmmcxw71_zigbee_ed_rx_off_bm_v1.axf• frdmmcxw71_zigbee_router_freertos_v1.axf• frdmmcxw71_zigbee_ed_rx_on_freertos_v1.axf• frdmmcxw71_zigbee_ed_rx_off_freertos_v1.axf
FRDM-MCXW72 board	• frdmmcxw72_zigbee_router_freertos_v1.axf• frdmmcxw72_zigbee_ed_rx_on_freertos_v1.axf• frdmmcxw72_zigbee_ed_rx_off_freertos_v1.axf
MCX-W71-EVK board	• mcxw71evk_zigbee_router_bm_v1.axf• mcxw71evk_zigbee_ed_rx_on_bm_v1.axf• mcxw71evk_zigbee_ed_rx_off_bm_v1.axf• mcxw71evk_zigbee_router_freertos_v1.axf• mcxw71evk_zigbee_ed_rx_on_freertos_v1.axf• mcxw71evk_zigbee_ed_rx_off_freertos_v1.axf
MCX-W72-EVK board	• mcxw72evk_zigbee_router_freertos_v1.axf• mcxw72evk_zigbee_ed_rx_on_freertos_v1.axf• mcxw72evk_zigbee_ed_rx_off_freertos_v1.axf

The table below shows that the OTA images are the V2/V3.ota files.

V2/V3 OTA files

Hardware platform	OTA files
FRDM-MCXW71 board	• frdmmcxw71_zigbee_router_bm_v2.ota• frdmmcxw71_zigbee_router_bm_v3.ota• frdmmcxw71_zigbee_ed_rx_on_bm_v2.ota• frdmmcxw71_zigbee_ed_rx_on_bm_v3.ota• frdmmcxw71_zigbee_ed_rx_off_bm_v2.ota• frdmmcxw71_zigbee_ed_rx_off_bm_v3.ota• frdmmcxw71_zigbee_router_freertos_v2.ota• frdmmcxw71_zigbee_router_freertos_v3.ota• frdmmcxw71_zigbee_ed_rx_on_freertos_v2.ota• frdmmcxw71_zigbee_ed_rx_on_freertos_v3.ota• frdmmcxw71_zigbee_ed_rx_off_freertos_v2.ota• frdmmcxw71_zigbee_ed_rx_off_freertos_v3.ota
FRDM-MCXW72 board	• frdmmcxw72_zigbee_router_freertos_v2.ota• frdmmcxw72_zigbee_router_freertos_v3.ota• frdmmcxw72_zigbee_ed_rx_on_freertos_v2.ota• frdmmcxw72_zigbee_ed_rx_on_freertos_v3.ota• frdmmcxw72_zigbee_ed_rx_off_freertos_v2.ota• frdmmcxw72_zigbee_ed_rx_off_freertos_v3.ota
MCX-W71-EVK board	• mcxw71evk_zigbee_router_bm_v2.ota• mcxw71evk_zigbee_router_bm_v3.ota• mcxw71evk_zigbee_ed_rx_on_bm_v2.ota• mcxw71evk_zigbee_ed_rx_on_bm_v3.ota• mcxw71evk_zigbee_ed_rx_off_bm_v2.ota• mcxw71evk_zigbee_ed_rx_off_bm_v3.ota• mcxw71evk_zigbee_router_freertos_v2.ota• mcxw71evk_zigbee_router_freertos_v3.ota• mcxw71evk_zigbee_ed_rx_on_freertos_v2.ota• mcxw71evk_zigbee_ed_rx_on_freertos_v3.ota• mcxw71evk_zigbee_ed_rx_off_freertos_v2.ota• mcxw71evk_zigbee_ed_rx_off_freertos_v3.ota
MCX-W72-EVK board	• mcxw72evk_zigbee_router_freertos_v2.ota• mcxw72evk_zigbee_router_freertos_v3.ota• mcxw72evk_zigbee_ed_rx_on_freertos_v2.ota• mcxw72evk_zigbee_ed_rx_on_freertos_v3.ota• mcxw72evk_zigbee_ed_rx_off_freertos_v2.ota• mcxw72evk_zigbee_ed_rx_off_freertos_v3.ota
Parent topic:*Zi over-the-air upgrade*	

OTA upgrade operation This section describes the OTA upgrade operation for FRDM-MCXW71, FRDM-MCXW72, MCX-W71-EVK or MCX-W72-EVK boards. For information about the over-the-air upgrade process for RW612 platforms, refer to the section, Zigbee Over The Air Upgrade (OTA) for RW612 platforms.

To add an image to the coordinator, the OTA images must be programmed. To program the OTA images, perform the following steps:

1. Use the J-Link utility.
2. Download J-Link from J-Link / J-Trace Downloads.
3. Plug the FRDM-MCXW71 or the FRDM-MCXW72 board, or the MCX-W71-EVK board, or the MCX-W72-EVK board to the USB port (no need to keep the **SW4** button pressed while doing this step).
4. Create a commands_script file with the following content (change the application name as necessary):

```
Reset
Halt
LoadBin <OTA_ADDRESS> 0x7A000
Reset
Go
Quit
```

Where OTA_ADDRESS value:

- for MCX-W71-EVK/FRDM-MCXW71 OTA_ADDRESS = 0x7A000
- for MCX-W72-EVK/FRDM-MCXW72 OTA_ADDRESS = 0xFA000

Note: If J-Link fails to recognize the .ota file, rename it to .bin and retry.

5. Copy the application and commands_script in the same folder where the J-Link executable is placed.
6. Run the following code:

```
JLink.exe/JLinkExe (linux) -<DEVICE_NAME> -if SWD -speed 4000 -autoconnect 1 -
↪CommanderScript commands_script
```

Where DEVICE_NAME can be:

- MCXW716 for MCXW71
- MCXW727C_CORE0 for MCXW72

When adding an image to a non-factory new coordinator, care must be taken not to use the Erase command to erase the flash.

Any devices with OTA clients in the network periodically send match descriptor requests to find an OTA server. Once a server responds, it then sends an IEEE address request to confirm the address details. The clients then periodically send OTA Image Requests to determine whether the server is hosting an image for that client device. In response to the Image Request, the server returns details of the image that it currently hosts: Manufacturer code, Image tag, and Version number. The client checks these credentials and decides whether it requires this image. If it does not, it queries the server again at the next query interval. If the client does require the image, it starts to issue Block Requests to the server to get the new image. Once all blocks of the new image have been requested and received, the new image is verified. The older image is invalidated, the device reboots, and runs the new image. The client resumes periodically querying the server for new images.

The End Device, which is RX Off, is allowed to enter Sleep mode. It stays awake for 5 seconds and then sleeps for 1 second when not performing "Finding and Binding".

Parent topic:*ZigBee over-the-air upgrade*

Image credentials Four main elements of the OTA header are used to identify the image to enable the OTA client to decide whether it must download the image.

- **Manufacturer code:** This element is a 16-bit number that is a ZigBee-assigned identifier for each member company. In this application, this number has been set to 0x1037, which is the identifier for NXP. In the final product, this number must be changed to the identifier of the manufacturer. The OTA client compares the Manufacturer code in the advertised image with its own and the image downloads only if they match.
- **Image type:** This element is a manufacturer-specific 16-bit number in the range 0x0000 to 0xFFBF. It is used by the manufacturer to distinguish between devices. In this application, the Image type is normally set to the ZigBee Device Type. However, this application uses 0x0003 for the Router and End Device. The OTA client compares the advertised Image type with its own Image type. If the Image type matches, then the image is downloaded. The product designers are entirely free to implement an identification scheme of their own.

- **File version:** This element is a 32-bit number representing the version of the image. The OTA client compares the advertised version with its current version before deciding whether to download the image.
- **OTA header string:** This element is a 32-byte character string and its use is manufacturer-specific. In this application, the OTA client compares the string in the advertised image with its own string before accepting an image for download. If the strings match, then the image is accepted. In this way, the string can be used to provide extra detail for identifying images, such as hardware subtypes.

Parent topic:*ZigBee over-the-air upgrade*

Upgrade and downgrade The decision to accept an image following a query response is under the control of the application. The code, as supplied, accepts an upgrade or a downgrade. As long as the notified image has the right credentials and a version number, which is different from the current version number, the image is downloaded.

For example, if a client is running a v3 image and a server is loaded with a v2 image then the v2 image is downloaded. The application callbacks the function responsible for handling the image in the following two scenarios:

- When the client is required to accept only the upgraded images (v2 > v3 > v5)
- When the client is required to accept only the sequential upgrade images (v2 > v3 > v4 > v5)

.

Parent topic:*ZigBee over-the-air upgrade*

LED indication table This section includes the LED states for the following device types:

- Coordinator
- Router
- End device RX on
- End device RX off (sleepy device)

Coordinator The table below lists the LED states for Coordinator.

LED1	LED2	NOTES
OFF	OFF	The device is not on the network
OFF	BLINKING ON/OFF every 500 ms	Network Steering/permit join is active
OFF	BLINKING ON/OFF every 1 second	Find and Bind initiated and is still active
OFF	ON	The device is active
OFF	BLINKING ON/OFF every 250 ms	Both Network Steering/permit join and find and bind are active

Parent topic:*LED indication table*

Router The table below lists the LED states for Router device type.

LED1	LED2	NOTES
OFF	OFF	The device is not on the network
OFF/ON (Current ON/OFF cluster status) or Blinking ON/OFF (Identifying)	Blinking ON/OFF every 250 ms	Network Steering/permit join is active
OFF/ON (Current ON/OFF cluster status) or Blinking ON/OFF (Identifying)	Blinking ON/OFF every 2 seconds	OTA aborted or failed
OFF/ON (Current ON/OFF cluster status) or Blinking ON/OFF (Identifying)	Blinking ON/OFF every 500 ms	OTA in progress

Parent topic:*LED indication table*

End Device RX On The table below lists the LED states for End Device RX On.

LED1	LED2	Remarks
OFF	OFF	The device is not on the network
OFF/ON (Current ON/OFF cluster status) or Blinking ON/OFF (Identifying)	Blinking ON/OFF every 250 ms	Find and Bind active
OFF/ON (Current ON/OFF cluster status) or BlinkingG ON/OFF (Identifying)	Blinking ON/OFF every 2 seconds	OTA aborted or failed
OFF/ON (Current ON/OFF cluster status) or Blinking ON/OFF (Identifying)	Blinking ON/OFF every 500 ms	OTA in progress

|

Parent topic:*LED indication table*

End Device RX Off (sleepy device) The table below lists the LED states for End device RX Off.

LED1	LED2	Remarks
OFF	OFF	The device is not on the network
OFF/ON (Current ON/OFF cluster status) or BLINKING ON/OFF (Identifying)	ON	The device is not sleeping and is active
BLINKING ON for 5 seconds and OFF for 1 second	BLINKING ON for 5 seconds and OFF for 1 second	The device is going through a sleep and wake cycle

Parent topic:*LED indication table*

Note about the source code in the document Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2023-2025 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials must be provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE

Revision history The table below summarizes revisions to this document.
Revision history

Document ID	Release date	Description
UG10194 v.4.2	24 March 2025	Added support for RW612 platform and moved to Markdown format.
UG10194 v.4.1	24 January 2025	Updated Allowing other devices to join the network (Router)
UG10194 v.4.0	29 November 2024	- Updated document ID to UG10194

ZigBee 3.0 Stack User Guide

Preface This manual provides a single point of reference for information relating to the ZigBee 3.0 wireless networking protocol and its associated stack, when implemented on the NXP K32W041, K32W061, K32W1, MCXW71, MCXW72, RW612, and JN518x family of wireless microcontrollers. It supports the NXP hardware platforms:- K32W148-EVK, FRDM-MCXW71, FRDM-MCXW72, MCX-W71-EVK, MCX-W72-EVK, FRDM-RW612, and RD-RW612-BGA. The manual provides both conceptual and practical information concerning the ZigBee 3.0 protocol and the supporting NXP software. Guidance is provided on the use of the NXP Application Programming Interfaces (APIs) for ZigBee 3.0. The API resources (such as functions, network parameters, enumerations, data types and events) are described in detail. The document can be used as a reference resource during ZigBee 3.0 application development.

Note:

1. The **ZigBee 3.0 protocol** employs the ZigBee PRO stack - in particular, ***Revision 22/ZigBee 2017*** of this stack. Therefore, this User Guide relates to this stack revision.
2. This User Guide is concerned with the development of applications that operate over the ZigBee PRO stack. These applications may conform to ZigBee 3.0 or may use ZigBee or manufacturer-specific application profiles. ZigBee 3.0 applications are based on ZigBee device types for which users should also refer to the ***ZigBee 3.0 Devices User Guide*** (***JNUG3131***).

For more detailed information on the ZigBee 3.0 standard, refer to the protocol specifications available from the ZigBee Alliance.

Organization of this manual This manual is divided into four parts:

- Part I: Concept and Operational Information consists of six chapters:
  - **Chapter 1:** *Preface* provides an overview of the document contents, conventions, support resources, and compatibility information.
  - **Chapter 2:** *ZigBee overview* introduces the ZigBee 3.0 wireless network protocol.
  - **Chapter 3:** *ZigBee PRO architecture and operation* describes the architecture and features of ZigBee 3.0.
  - **Chapter 4:** *ZigBee Stack Software* introduces the NXP ZigBee PRO stack software.
  - **Chapter 5:** *Application development overview* provides an overview of the ZigBee 3.0 application development environment and process.
  - **Chapter 6:** *Application coding with ZigBee PRO APIs* describes how to perform common wireless network operations using the functions of the NXP ZigBee 3.0 APIs.
- Part II: Reference Information consists of six chapters:
  - **Chapter 7:** *ZigBee Device Objects (ZDO) API* details the functions and associated resources of the ZigBee Device Objects (ZDO) API.
  - **Chapter 8:** *Application Framework (AF) API* details the functions and associated resources of the Application Framework (AF) API.
  - **Chapter 9:** *ZigBee Device Profile (ZDP) API* details the functions and associated resources of the ZigBee Device Profile (ZDP) API.
  - **Chapter 10:** *General ZPS Resources* details the general functions and associated resources provided with the NXP ZigBee PRO stack.
  - **Chapter 11:** *Event and Status Codes* details the stack events and the return/status codes used by the ZigBee PRO APIs.
  - **Chapter 12:** *ZigBee network parameters* details the ZigBee network parameters.
- Part III: Network Configuration consists of one chapter:
  - **Chapter 13:** *ZPS Configuration Editor* describes how to use the ZPS Configuration Editor.
- Part IV: contains various ancillary information that include the following:
  - **Appendix A:** *Appendix A: Handling stack events*: a description of the handling of ZigBee PRO stack events.
  - **Appendix B:** *Appendix B: Application design notes*: a set of design application notes.
  - **Appendix C:** *Appendix C: Implementation of frame counters*: a description of frame counters.
  - **Appendix D:** *Appendix D: storing applications in device flash memory*: a description of application storage in Flash memory.
  - **Appendix E:** a glossary of terms used in ZigBee 3.0 networks.
  - **Appendix F:** Revision History for the document.

Parent topic:*Preface*

Conventions Files, folders, functions and parameter types are represented in **bold** type. Function parameters are represented in *italics*type.

Code fragments are represented in the Courier Newtypeface.

Note: A note represents an important fact, guideline, or information.

Parent topic:*Preface*

Acronyms and abbreviations The table below lists the acronyms and abbreviations used in this document.

Acronym	Description
AF	Application Framework
AIB	APS Information Base
APDU	Application Protocol Data Unit
API	Application Programming Interface
APS	Application Support (sub-layer)
APSDE	Application Support (sub-layer) Data
APSME	Application Support (sub-layer) Management
BDB	Base Device Behavior
DIO	Digital Input/Output
EPID	Extended PAN ID
HA	Home Automation
HVAC	Heating, Ventilation, and Air-Conditioning
IO	Input/Output
ISR	Interrupt Service Routine
MAC	Media Access Control
PAN	Personal Area Network
NIB	NWK Information Base
NPDU	Network Protocol Data Unit
NVM	Non-Volatile Memory Manager
NWK	Network
OS	Operating System
PDU	Protocol Data Unit
PDUM	Protocol Data Unit Manager
PIC	Programmable Interrupt Controller
RF	Radio Frequency
SAP	Service Access Point
SDK	Software Developer's Kit
UART	Universal Asynchronous Receiver-Transmitter
ZCL	ZigBee Cluster Library
ZCP	ZigBee Compliant Platform
ZDO	ZigBee Device Objects
ZDP	ZigBee Device Profile
ZCP	ZigBee Compliant Platform
ZDO	ZigBee Device Objects
ZDP	ZigBee Device Profile

Parent topic:*Preface*

Related documents For further information, refer to the following documents:

- JNUG3131 ZigBee 3.0 Devices User Guide
- JNUG3132 ZigBee Cluster Library (for ZigBee 3.0) User Guide
- JNUG133 Core Utilities User Guide
- JNUG3134 ZigBee Green Power (for ZigBee 3.0) User Guide

Parent topic:*Preface*

Support resources

- To access online support resources such as SDKs, Application Notes, and User Guides, visit the Wireless Connectivity page of the NXP website: https://www.nxp.com/products/wireless:WIRELESS-CONNECTIVITY
- ZigBee resources can be accessed from the ZigBee page, whose URL is: https://www.nxp.com/pages/jn516x-7x-zigbee-3-0:ZIGBEE-3-0.

All NXP resources referred to in this manual can be found at the above addresses, unless otherwise stated.

Parent topic:*Preface*

Trademarks All trademarks are the property of their respective owners.

Parent topic:*Preface*

Chip compatibility The software described in this manual can be used on the NXP K32W041, K32W061, K32W1, MCXW71, MCXW72, RW612, and JN518x family of wireless microcontrollers.

Parent topic:*Preface*

ZigBee overview The ZigBee protocol was developed to provide low-power, wireless connectivity for a wide range of network applications concerned with monitoring and control. ZigBee is a worldwide open standard controlled by the ZigBee Alliance. ZigBee PRO was then developed as an enhancement of the original ZigBee protocol, providing a number of extra features that are particularly useful for very large networks (that may include hundreds or even thousands of nodes).

ZigBee features The ZigBee standard builds on the established IEEE 802.15.4 standard for packet-based wireless transport. ZigBee enhances the functionality of IEEE 802.15.4 by providing flexible, extendable network topologies with integrated set-up and routing intelligence to facilitate easy installation and high resilience to failure. ZigBee networks also incorporate listen-before-talk and rigorous security measures that enable them to co-exist with other wireless technologies (such as Bluetooth and Wi-Fi) in the same operating environment.

ZigBee provides wireless connectivity that enables it to be installed on networks easily and cheaply. Its built-in intelligence and flexibility allow networks to be easily adapted to changing needs by adding, removing, or moving network nodes. The protocol is designed in such a manner that nodes can appear in and disappear from the network. Thus, it allows some devices to be put into a power-saving mode, when not active. This feature allows many devices in a ZigBee network to be battery-powered, making them self-contained and reduces installation costs.

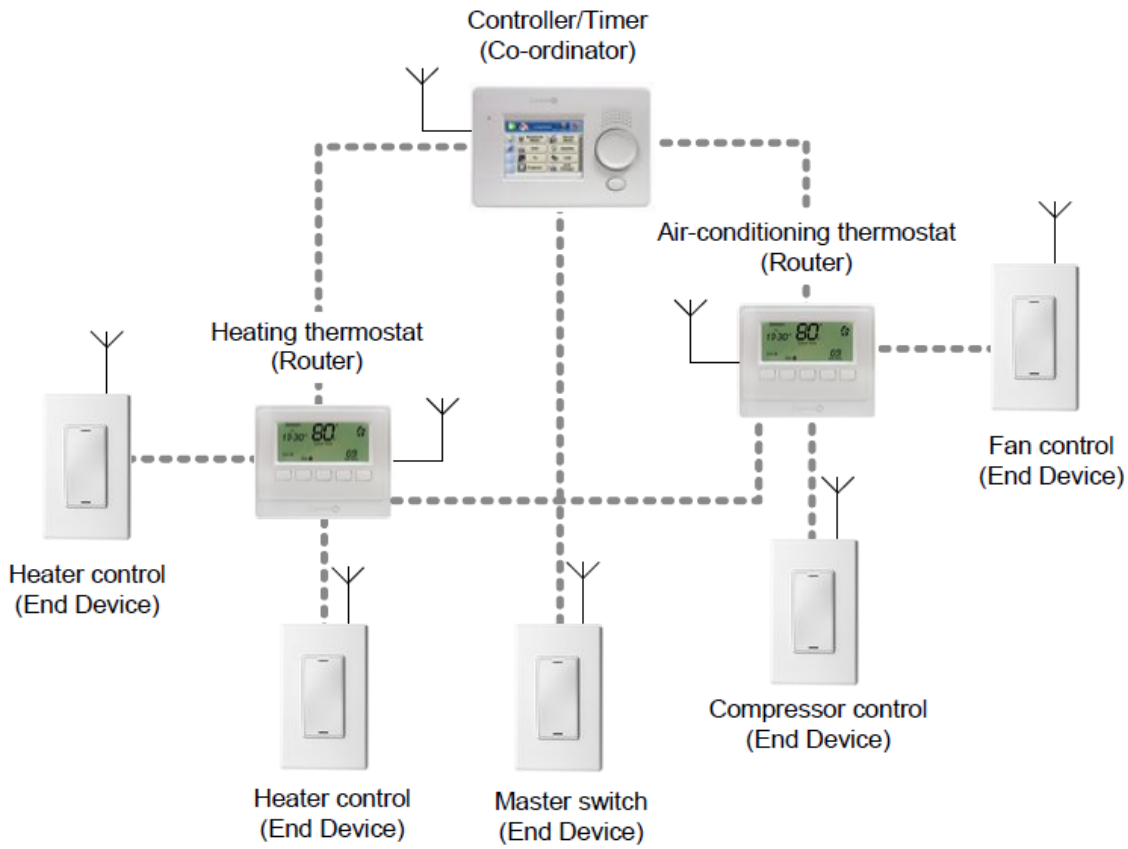
Figure 1 shows a simple example of a ZigBee network in an HVAC (Heating, Ventilation, and Air-Conditioning) system.

Parent topic:*ZigBee overview*

ZigBee 3.0 ZigBee 3.0 employs the ZigBee PRO protocol and is designed to facilitate general wireless networks that are not market-specific. Thus, devices from different market sectors can belong to the same wireless network. For example, lighting and healthcare devices in a hospital may share a single ZigBee network, allowing data to be routed through any intermediate (routing) device, irrespective of the device functionality.

Connecting the network to the Internet brings the devices into the 'Internet of Things' (IoT), allowing the network devices to be controlled and monitored from IP-based devices such as computers, tablets, and smartphones.

Simple ZigBee Network (Heating and Air-conditioning)



Parent topic:*ZigBee overview*

ZigBee network nodes A wireless network consists of a set of nodes that can communicate with each other by means of radio transmissions, according to a set of routing rules (for passing messages between nodes). A ZigBee wireless network includes three types of node:

- **Coordinator:** This is the first node to be started and is responsible for forming the network by allowing other nodes to join the network through it. Once the network is established, the Coordinator has a routing role (is able to relay messages from one node to another) and is also able to send/receive data. Every network must have one and only one Coordinator.
- **Router:** This is a node with a routing capability, and is also able to send/receive data. It also allows other nodes to join the network through it, so plays a role in extending the network. A network may have many Routers.
- **End Device:** This is a node which is only capable of sending and receiving data (it has no routing capability). A network may have many End Devices.

The ZigBee PRO network topology describes deployment of these node types in a ZigBee PRO network. More detailed information about the node types is provided in ZigBee nodes.

Parent topic:*ZigBee overview*

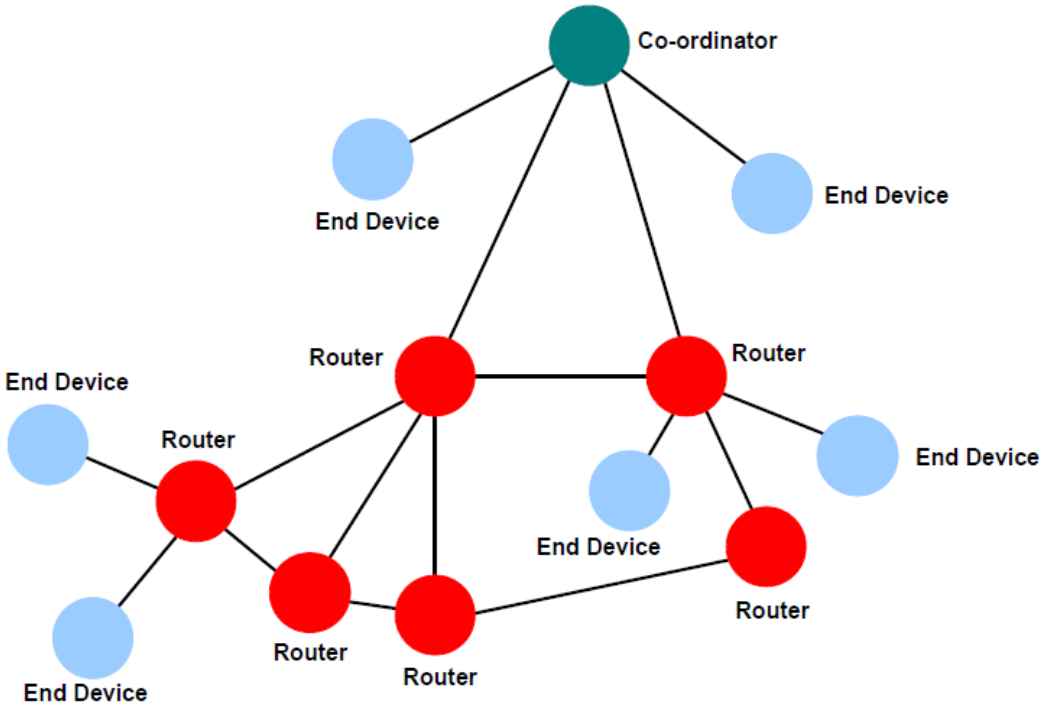
ZigBee PRO network topology ZigBee facilitates a range of network topologies from the simplest Star topology, through the highly structured Tree topology to the flexible Mesh topology. ZigBee PRO is designed primarily for Mesh networks.

A Mesh network has little implicit structure. It is a collection of nodes comprising a Coordinator and a number of Routers and/or End Devices, where:

- Each node, except the Coordinator, is associated with a Router or the Co- ordinator - this is the node through which it joined the network and is known as its 'parent'. Each parent may have a number of 'children'.
- An End Device can only communicate directly with its own parent.
- Each Router and the Coordinator can communicate directly with any other Router/Coordinator within radio range.

It is the last property above that gives a Mesh network its flexibility and efficiency in terms of inter-node communication. A Mesh network is illustrated in the figure below.

Simple Mesh Network



Parent topic:*ZigBee overview*

Ideal applications for ZigBee ZigBee is suitable for a wide range of applications, covering both commercial and domestic use. These applications include:

- Point-to-point cable replacement (for example, wireless mouse, remote controls, toys)
- Security systems (for example, fire and intruder)
- Environmental control (for example, heating and air-conditioning)
- Hospital patient monitoring
- Lighting control
- Home automation (for example, home entertainment, doors, gates, curtains, and blinds)
- Automated meter reading (AMR)
- Industrial automation (for example, plant monitoring and control)

ZigBee provides wireless communication that enables those applications to be developed, which currently cannot be implemented with cabled systems. Examples are applications that involve

mobility, which must be free of cabling (such as long-term health monitoring, asset tracking in warehouses). Existing applications (such as lighting control and industrial plant monitoring) that currently rely on cable-based systems can be implemented more cheaply as ZigBee reduces or removes cable installation costs. ZigBee can also be beneficial in environments where cable-based solutions are difficult and expensive to install. Home security systems are examples of such systems. In these systems, the sensors should be easy to install (no cables or power supply wiring) and small and self-contained (battery-powered).

Parent topic:*ZigBee overview*

Wireless radio frequency operation The IEEE 802.15.4 protocol, on which ZigBee is built, provides radio-based network connectivity operating in one of three possible RF (Radio Frequency) bands: 868 MHz, 915 MHz, or 2400 MHz. These bands are available for unlicensed use, depending on the geographical area (check your local radio communication regulations).

The characteristics of these RF bands are shown in the table below.

Total number of channels

RF Band	Number of Channels
863 MHz - 876 MHz	63
915 MHz - 921 MHz	27
Total	90

Channel distribution across pages

Channel Page	Description
863-876 MHz	63
915-921 MHz	27
Total	90

The internal representation of the channels in our stack is as follows:

- A 32-bit mask is used to represent the channel mask.
- The top 5 bits are used for page number and the lower 27 bits are the channel masks.

In 2.4G, page number is 0 channel range 11-26. Thus, it will be 0x00000800 (page 0, Channel 11). In Sub Gig Page 28 channel 0, is 0xE0000001. The 868 MHz and 915 MHz bands offer certain advantages such as fewer users, less interference, and less absorption and reflection, but the 2400 MHz band is far more widely adopted for a number of reasons:

- Worldwide availability for unlicensed use
- Higher data rate (250 kbit/s) and more channels
- Lower power (transmit/receive are on for shorter time due to higher data rate)
- Band more commonly understood and accepted by the marketplace

Therefore, the ZigBee standard assumes operation in the 2400-MHz band, although it is possible to implement ZigBee networks in the other IEEE 802.15.4 bands. ZigBee includes measures to avoid interference between radio communications. One is its ability to automatically select the best frequency channel at initialization. It is also possible to adapt to a changing RF environment by moving the network to another channel, if the current channel proves problematic - this 'frequency agility' is a core feature of ZigBee PRO. Other measures are described in Highly reliable operation. The range of a radio transmission is dependent on the operating environment - for example, indoors or outdoors. Using an NXP JN518x or K32W041/K32W061/K32W1/MCXW71/MCXW72/RW612 standard module fitted with an external dipole antenna, a range of over 1 km can typically be achieved in an open area, but inside a

building this can be reduced due to absorption, reflection, diffraction and standing wave effects caused by walls and other solid objects. A high-power module (greater than 15 dBm output power) can achieve a range which is a factor of five greater than that of a standard module. In addition, the range between devices can be extended in a ZigBee network since the network topology (see Network topology) can use intermediate nodes (Routers) as stepping stones when passing data to destinations.

Parent topic:*ZigBee overview*

Battery-powered components There are many wireless applications that benefit from battery power, including light-switches, active tags and security detectors. The ZigBee and IEEE 802.15.4 protocols are specifically designed for battery-powered applications. From a user perspective, battery power has certain advantages:

- Easy and low-cost installation of nodes:No need to connect node to separate power supply.
- Flexible location of nodes:Nodes can be installed in difficult places where there is no power supply, and can even be used as mobile devices.
- Easily modified network:Nodes can easily be added or removed, on a temporary or permanent basis.

Since these devices are generally small, they use low-capacity batteries and therefore battery use must be optimized. This is achieved by restricting the amount of time for which energy is required by the device.

- Since the major power drain in the system is the operation of the radio, data may be transmitted infrequently (perhaps once per hour or even once per week), which results in a low duty cycle (transmission time as proportion of time interval between transmissions).
- When data is not being sent, the device may revert to a low-power 'sleep' mode to minimize power consumption.

In practice, not all nodes on a network can be battery-powered, notably those that need to be switched on all the time for routing purposes (and therefore cannot sleep). These devices can often be installed in a mains-powered appliance that is permanently connected to the mains supply (even if not switched on) - for example, a ceiling lamp or an electric radiator. This avoids the need to install a dedicated mains power connection for the node. Only End Devices are normally battery-powered.

Note:

A network device can also potentially use "energy harvesting" to absorb and store energy from its surroundings - for example, the use of a solar cell panel on a device in a well-lit environment.

Parent topic:*ZigBee overview*

Easy installation and configuration One of the great advantages of a ZigBee network is the ease with which it can be installed and configured.

As already mentioned, the installation is simplified and streamlined by the use of certain battery-powered devices with no need for power cabling. In addition, since the whole system is radio-based, there is no need for control wiring to any of the network devices. Therefore, ZigBee avoids much of the wiring and associated construction work required when installing cable-based networks.

The configuration of the network depends on how the installed system has been developed. There are three system possibilities: pre-configured, self-configuring, and custom.

- **Pre-configured system:** A system in which all parameters are configured by the manufacturer. The system is used as delivered and cannot readily be modified or extended. Examples: vending machine, patient monitoring unit.

- **Self-configuring system:** A system that is installed and configured by the end-user. The network is initially configured by sending "discovery" messages between devices. Some initial user intervention is required to set up the devices - for example, by pressing buttons on the nodes. Once installed, the system can be easily modified or extended without any re-configuration by the user. The system detects when a node has been added, removed, or simply moved, and automatically adjusts the system settings.

  Example: off-the-shelf home security or home lighting system in which extra devices can be added later.

- **Custom system:** A system that is adapted for a specific application/location. It is designed and installed by a system integrator using custom network devices. The system is usually configured using a software tool.

As indicated above, system commissioning (individually configuring the network nodes) can be performed in either of the below modes:

- By using an IO interface (for example, buttons or a keypad) on the node in a self-configuring system.
- By using a commissioning tool (for example, by running on a lap-top PC) that interacts with the node in a custom system.

In the latter case, ZigBee PRO allows commissioning to be conducted in a secure way - for example, using a security key to gain access to the configurable parameters of the node, and using encryption in any wireless communication between the commissioning tool and the node. For more information on system security, refer to Secure operating environment.

Parent topic:*ZigBee overview*

Highly reliable operation ZigBee and IEEE 802.15.4 employ a range of techniques to ensure reliable communications between network nodes - that is, to ensure communications reach their destinations uncorrupted. Corruption could result, for example, from radio interference or poor transmission/reception conditions.

- **Data Coding:** At a first level, a coding mechanism is applied to radio transmissions. The coding method employed in the 2400-MHz band uses QPSK (Quadrature Phase-Shift Keying) modulation with conversion of 4-bit data symbols to 32-bit chip sequences. This coding results in a high probability that a message reaches its destination intact, even if there are conflicting transmissions. (A conflicting transmission implies that more than one device transmits in the same frequency channel at the same time).
- **Listen Before Send:** The transmission scheme also avoids transmitting data when there is activity on its chosen channel - this is known as Carrier Sense, Multiple Access with Collision Avoidance (CSMA-CA). Put simply, this means that before beginning a transmission, a node listens on the channel to check whether it is clear. If activity is detected on the channel, the node delays the transmission for a random amount of time and listens again - if the channel is now clear, the transmission can begin, otherwise the delay-and-listen cycle is repeated.
- **acknowledgments:** Two systems of acknowledgments are available to ensure that messages reach their destinations:
  - **End-to-End:** When a message arrives at its final destination, the receiving device sends an acknowledgment to the source node to indicate that the message has been received. End-to-end acknowledgments are optional.
  - **Next Hop:** When a message is routed via intermediate nodes to reach its destination, the next routing node (or 'next hop' node) in the route sends an acknowledgment to the previous node to indicate that it has received the message. Next-hop acknowledgments are always implemented.

In both cases, if the sending device does not receive an acknowledgment within a certain time interval, it resends the original message (it can resend the message several times until the message has been acknowledged).

- **Frequency Agility:** When a ZigBee network is initially set up, the 'best' channel in the relevant radio band is automatically chosen as the operating channel. The operating channel is normally the quietest channel detected in an energy scan across the band. However, it might not always remain the quietest channel if other networks that operate in the same channel are introduced nearby. For this reason, ZigBee includes an optional frequency agility facility. If the operating channel becomes too noisy, this feature allows the whole network to be moved to a better channel in the radio band.
- **Route Repair:** Networks that employ a Mesh topology (see ZigBee PRO network topology) have built-in intelligence to ensure that messages reach their destinations. If the default route to the destination node is down, due to a failed intermediate node or link, the network can 'discover' and implement alternative routes for message delivery. ZigBee PRO is designed for Mesh networks and therefore incorporates "route repair" as a core feature.

The above reliability measures allow a ZigBee network to operate even when there are other ZigBee networks nearby operating in the same frequency band. Therefore, adjacent ZigBee networks do not interfere with each other. In addition, ZigBee networks can also operate in the neighborhood of networks based on other standards, such as Wi-Fi and Bluetooth, without any interference.

Parent topic:*ZigBee overview*

Secure operating environment ZigBee networks can be made secure - measures can be incorporated to prevent intrusion from potentially hostile parties and from neighboring ZigBee networks. ZigBee also provides privacy for communication between nodes of the same network.

ZigBee PRO security includes the following features:

- Access control lists
- Key-based encryption of communications
- Frame counters

These security measures are outlined below.

Access control lists An access control list allows only pre-defined 'friendly' nodes to join the network.

Parent topic: Secure operating environment

Key-based encryption A very high-security, 128-bit AES-based encryption system (built into the device as a hardware function) is applied to network communications, preventing external agents from interpreting ZigBee network data.

This encryption is key-based. Normally, the same 'network key' is used for all nodes in the network. However, it is possible to use an individual 'link key' between a given pair of network nodes, allowing communications (possibly containing sensitive data) between the two nodes to be private from other nodes in the same network.

Keys can be pre-configured in nodes in the factory, commissioned during system installation or distributed around a working network from a central 'Trust Centre' node. A Trust Centre manages keys and security policies - for example, changing the network key on all network nodes, issuing link keys for node pairs and restricting the hours in which certain events or interactions can occur. Any node can be nominated as the Trust Centre, but it is by default the Coordinator.

A distributed security model can alternatively be used, which does not have a Trust Centre - instead, security is managed by the Router nodes in the network.

Parent topic: Secure operating environment

Frame counters The use of frame counters prevents sending the same message twice, and freshness checking rejects any such repeated messages, preventing message replay attacks on the network. An example of a replay attack would be someone recording the open command for a garage door opener, and then replaying it to gain unauthorized entry into the property. Frame counters are described in more detail in the Appendix A, *Appendix C: Implementation of frame counters*.

Parent topic: Secure operating environment

Parent topic:*ZigBee overview*

Co-existence and interoperability ZigBee is an open standard devised by the ZigBee Alliance. Any device designed for use in a ZigBee network must comply with the standard. This ensures "co-existence" and, to a certain extent, "interoperability" of ZigBee devices:

- **Co-existence:** The ability of a device to operate in the same space and radio channel as devices in other wireless networks (which possibly use protocols other than ZigBee) without interfering with them
- **Interoperability:** The ability of a device to operate in the same ZigBee network as devices from other manufacturers - that is, to communicate and function with them.

The ZigBee Alliance coordinates the compliance issues for products based on the ZigBee protocol. It defines two levels of compliance:

- **ZigBee Compliant Platform (ZCP)** applies to modules or platforms intended as building blocks for use in end-products. All NXP products based on the supported chips are designed to be ZigBee Compliant Platforms. Refer to the section Chip compatibility.
- **ZigBee Certified Product** applies to end-products that are built on ZigBee Compliant Platforms, and that use public ZigBee Alliance device types and clusters. After successful completion of the ZigBee Alliance Certification program, the ZigBee Certified Product logo can be applied to the product.

  Note: End-products based on manufacturer-specific device types and clusters can also obtain ZigBee Certified Product status, but such products cannot carry the ZigBee Certified Product logo.

Test service providers are authorized by the ZigBee Alliance to undertake testing and certification. For details of authorized test houses, contact the ZigBee Alliance.

In addition, products using an NXP ZCP must also be checked against the radio regulations of the country or countries where they are to be marketed (these checks can often be performed by the same test house).

Parent topic:*ZigBee overview*

Device types and clusters For the purpose of interoperability (described in Section 2.9), the ZigBee Alliance employs the concepts of a device type and a cluster, which define the functionality of a network node. Clusters and device types are introduced below (but more detailed information can be found in Section 3.4).

Note: The ZigBee 'application profile' (which collects together the device types for a market sector) is not so

prevalent in ZigBee 3.0. However, application profiles are still supported for backward compatibility.

Clusters A cluster is a software entity that encompasses a particular piece of functionality for a network node. A cluster is defined by a set of attributes (parameters) that relate to the functionality and a set of commands (that can typically be used to request operations on the cluster attributes). As an example, a thermostat uses the Temperature Measurement cluster that

includes attributes such as the current temperature measurement, the maximum temperature that can be measured, and the minimum temperature that can be measured. However, the only operations that needs to be performed on these attributes would be reads and writes.

The ZigBee Alliance defines a collection of clusters in the ZigBee Cluster Library (ZCL). These clusters cover the functionalities that are most likely to be used. The NXP implementations of these clusters are provided in the ZigBee 3.0 Software Developer's Kit (SDK) and are described in the *ZigBee Cluster Library User Guide (JNUG3132)*.

Parent topic: Device types and clusters

Device types The complete functionality of a network node is determined by its device type. This defines a collection of clusters (some mandatory and some optional) that make up the supported features of the device. For example, the Thermostat device uses the Basic and Temperature Measurement clusters, and can also use one or more optional clusters. A device is an instance of a device type.

A network node can support more than one device type. The application for a device type runs on a software entity called an endpoint and each node can have up to 240 endpoints.

All ZigBee 3.0 nodes must implement the ZigBee Base Device (which does not occupy an endpoint), which handles fundamental operations such as commissioning.

The ZigBee device types and ZigBee Base Device are detailed in the *ZigBee Devices User Guide (JNUG3131)*.

Parent topic: Device types and clusters

Parent topic:*ZigBee overview*

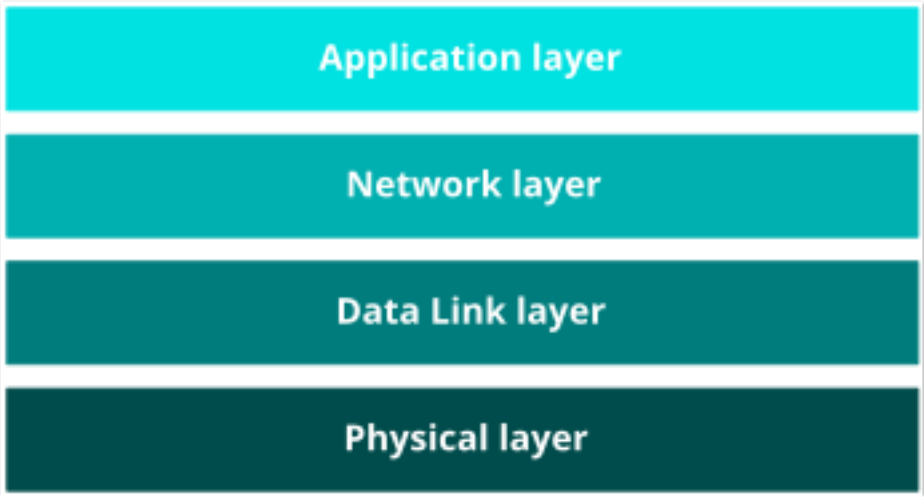
ZigBee PRO architecture and operation This chapter introduces ZigBee PRO from architectural and operational view-points by describing:

- Basic architecture on which ZigBee PRO is based. See Section 3.1, (Architectural overview)
- Concepts for an understanding of ZigBee PRO at the network level. See Section 3.2 (Network level concepts)
- Process of network formation. See Section 3.3, (Network creation)
- Concepts for an understanding of ZigBee PRO at the application level. See Section 3.4, (Application level concepts)
- Features and concepts related to message routing. See Section 3.5, (Network routing)
- Features and concepts related to exchanging messages. See Section 3.6, (Network communications)
- A detailed view of the ZigBee PRO software architecture. See Section 3.7, (Detailed architecture)

Architectural overview This section introduces the basic architecture of the software that runs on a ZigBee PRO network node. The software architecture is built on top of IEEE 802.15.4, an established and proven standard for wireless communication.

From a high-level view, the software architecture of any ZigBee network consists of four basic stack layers: Application layer, Network layer, Data Link layer and Physical layer. The Application layer is the highest level and the Physical layer is the lowest level, as illustrated in the figure below.

Basic software architecture



The basic layers of the ZigBee software stack are described below, from top to bottom:

- Application layer: The Application layer contains the applications that run on the network node. These give the device its functionality - essentially an application converts input into digital data, and/or converts digital data into output. A single node may run several applications - for example, an environmental sensor may contain separate applications to measure temperature, humidity and atmospheric pressure.
- Network layer: The Network layer provides the ZigBee PRO functionality and the application's interface to the IEEE 802.15.4 layers. The layer is concerned with network structure and multi-hop routing.
- Data Link layer: The Data Link layer is provided by the IEEE 802.15.4 standard and is responsible for addressing - for outgoing data it determines where the data is going, and for incoming data it determines where the data has come from. It is also responsible for assembling data packets or frames to be transmitted and disassembling received frames. In the IEEE 802.15.4 standard, the Data Link layer is referred to as IEEE 802.15.4 MAC (Media Access Control) and the frames used are MAC frames.
- Physical layer: The Physical layer is provided by the IEEE 802.15.4 standard and is concerned with the interface to the physical transmission medium (radio, in this case). It is facilitates exchange of data bits with this medium, as well as with the layer above (the Data Link layer). In the IEEE 802.15.4 standard, the Physical layer is referred to as IEEE 802.15.4 PHY.

For a more detailed view of the software architecture of ZigBee PRO, refer to Section 3.7, Detailed architecture.

Note: Security measures are implemented throughout the stack, including the Application layer and lower stack layers.

Parent topic:*ZigBee PRO architecture and operation*

Network level concepts This section describes important concepts relating to the work of the ZigBee stack.

ZigBee nodes There are three general types of node that can exist in a ZigBee network:

- Coordinator
- Router
- End Device

Note: These roles exist at the network level - a ZigBee node may also be performing tasks at the Application level, independent of the role it plays in the network.

For example, a network of ZigBee devices measuring temperature may have a temperature sensor application in each node, irrespective of whether the node is an End Device, Router, or the Coordinator.

The roles of these node types are described in the sub-sections below.

Coordinator All ZigBee networks must have one (and only one) Coordinator.

At the network level, the Coordinator is mainly needed at system initialization - it is the first node to be started and performs the following initialization tasks:

- Selects the frequency channel to be used by the network (usually the one with the least detected activity)
- Starts the network
- Allows child nodes to join the network through it

The Coordinator can additionally provide other services such as message routing and security management. It may also provide services at the Application level. If any of these additional services are used, the Coordinator must be able to provide them at all times. However, if none of these additional services are used, the network will be able to operate normally even if the Coordinator fails or is switched off.

Parent topic: ZigBee nodes

Router A ZigBee PRO network usually has at least one Router. The main tasks of a Router are:

- Relays messages from one node to another.
- Allows child nodes to join the network through it.

Note: An important feature of the Router is that it cannot sleep, as it must always be available for routing.

Parent topic: ZigBee nodes

End Device The main task of an End Device at the network level is sending and receiving messages. An End Device can only communicate directly with its parent, so all messages to/from an End Device pass via its parent.

An End Device can be battery-powered and, when not transmitting or receiving, can sleep in order to conserve power. The parent device buffers messages destined for a sleep-enabled End Device. The End Device collects these messages once it is awake (also see Section 3.2.2 Network topology below).

Note: End Devices cannot relay messages and cannot allow other nodes to connect to the network through them. In other words, it implies that they cannot have children.

Parent topic: ZigBee nodes

Parent topic: Network level concepts

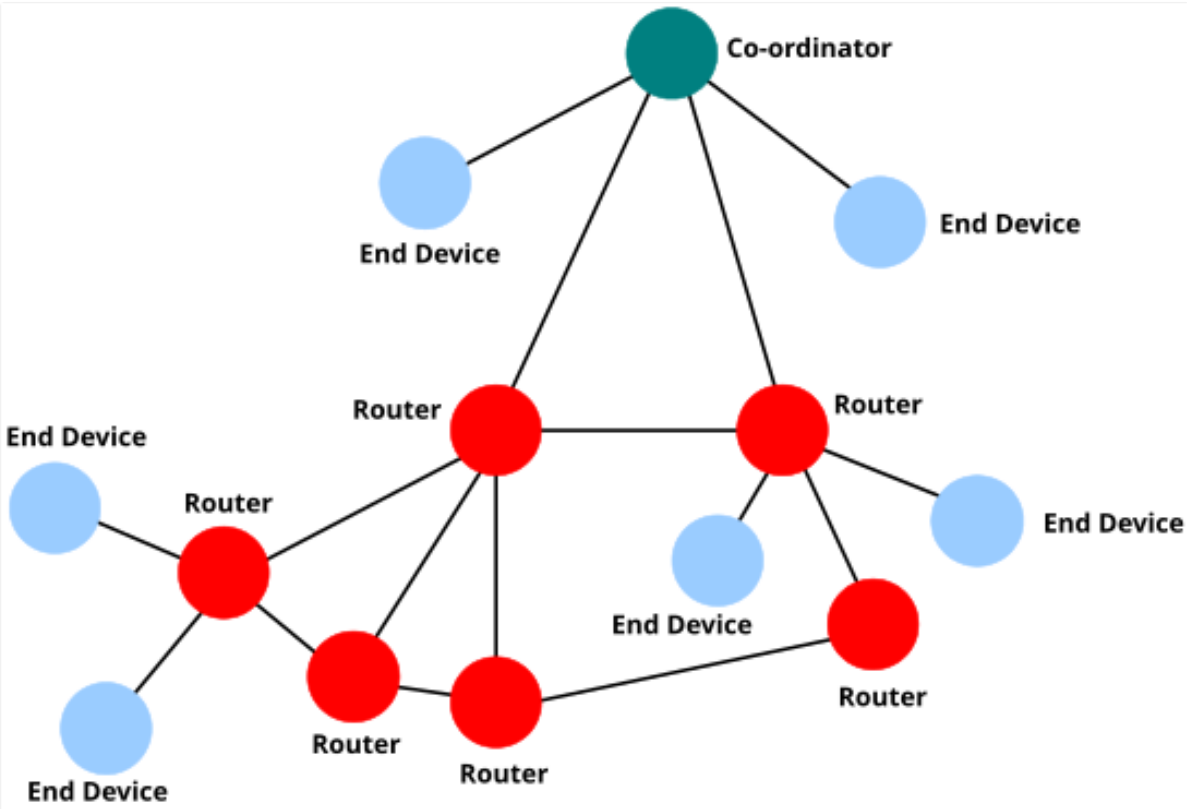
Network topology The ZigBee PRO standard was designed to facilitate wireless networks with the Mesh topology.

A Mesh network consists of a Coordinator, Routers, and End Devices. The Coordinator is associated with a set of Routers and End Devices - its children. A Router may then be associated with more Routers and End Devices - its children. This can continue to a number of levels. The relationships between the nodes must obey the following rules:

- The Coordinator and Routers can have children, and can therefore be parents.
- A Router can be both a child and a parent.
- End Devices cannot have children, and therefore cannot be parents. The communication rules for a Mesh network are as follows:
- An End Device can only directly communicate with its parent (and with no other node).
- A Router can directly communicate with its children, with its own parent, and with any other Router or Coordinator within radio range.
- The Coordinator can directly communicate with its children and with any Router within radio range.

The resulting structure is illustrated in the figure below.

Mesh topology



In ZigBee PRO, the maximum depth (number of levels below the Coordinator) of a network is 15. The maximum number of hops that a message can make in traveling between the source and destination nodes is 30 (twice the maximum depth).

A routing node (Router or Coordinator) can communicate directly with other routing nodes within radio range. This specific property distinguishes a Mesh network from a Tree network. This property enables very efficient and flexible message propagation. It also implies that alternative routes can be found if a link fails or there is congestion.

Note: An End Device, which is able to sleep, is unable to receive messages directly. A message destined for a sleep-enabled End Device is always buffered in its parent node if the End Device is asleep when the message arrives. Once the End Device is awake, it must ask or 'poll' the parent for messages. In the Mesh topology, a 'route discovery' feature is provided, which allows the network to find the best available route for a message. Refer further details in Section 3.5.2, "**Route discovery**".

Note: Message propagation is handled by the network layer software and is transparent to the application programs running on the nodes.

Parent topic: Network level concepts

Neighbor tables A routing node (Router or Coordinator) holds information about its neighboring nodes. This information is stored in a Neighbor table containing entries for the node's immediate children, for its own parent and, in a Mesh network, for all peer Routers with which the node has direct radio communication.

It is possible to define the maximum number of entries in a Neighbor table. If this parameter is set to a low value, it will result in a 'long, thin network'.

The structure and configuration of a Neighbor table are described in Appendix B.5.1.

Parent topic: Network level concepts

Network addressing In a ZigBee network, each node must have a unique identification. For this purpose, each node has two addresses:

- **IEEE (MAC) address**: A 64-bit address, allocated by the IEEE, which uniquely identifies the device. No two devices in the world can have the same IEEE address. It is often referred to as the MAC address. In a ZigBee network, it is sometimes called the 'extended' address.
- **Network address:** A 16-bit address that identifies the node in the network and is local to that network. Thus, two nodes in separate networks may have the same network address. It is sometimes called the 'short' address.

In ZigBee PRO, the network address of a node is dynamically assigned as a random 16-bit value by the parent when the node first joins the network. This is known as stochastic addressing due to the randomness of the address allocation. Although random, the parent ensures that the chosen address has not already been assigned to one of its neighbors. In the unlikely event of the address already existing in the network beyond the immediate neighborhood, a mechanism exists to automatically detect and resolve the conflict. The allocated network address can be retained by the joining node, even if it later loses its parent and acquires a new parent.

The Coordinator always has the network address 0x0000.

While an application on a node may use IEEE/MAC addresses or network addresses to identify remote nodes, the ZigBee PRO stack always uses network addresses for this purpose. To facilitate translation between IEEE/MAC addresses and network addresses, an Address Map table may be maintained on the node, where each table entry contains the pair of addresses for a remote node.

In the NXP implementation of ZigBee PRO, the IEEE/MAC addresses (of other network nodes) are stored in a single place on a node, called the MAC Address table. This avoids the need to repeat the 64-bit IEEE/MAC addresses in other tables, such as the Address Map table and Neighbor table, and therefore saves storage space. Instead, a 16-bit index to the relevant entry in the MAC Address table is stored in the other tables.

It is also possible to define a 16-bit 'group address' which refers to a set of applications (or endpoints that may be located across several nodes. For details, refer to Section 3.4.1, Multiple applications and endpoints).

Specifying a group address in a data transfer results in the data being broadcast to all nodes in the network but, at the destinations, the data is only passed to those applications, which are covered by the group address. Refer to Section 6.3, Managing group addresses for more details of using group addresses.

Parent topic: Network level concepts

Network identity A ZigBee network must be uniquely identifiable. This allows more than one ZigBee network to operate in close proximity - nodes operating in the same space must be able to identify which network they belong to.

For this purpose, ZigBee uses two identifiers, as follows:

- **PAN ID:** A 16-bit value called the PAN ID (Personal Area Network Identifier) is used in inter-node communications (implemented at the IEEE 802.15.4 level of the stack) to identify the relevant network. A value for the PAN ID is selected at random by the Coordinator when the network is started. When other nodes join the network, they learn the network's PAN ID and use it in all subsequent communications with the network.

It is possible that the PAN ID generated for a newly installed network clashes with the PAN ID of another network already operating on the same radio channel, in the same neighborhood. In this case, ZigBee PRO automatically resolves such a conflict by generating another random PAN ID for the new network. This continues until a value is obtained that does not clash with the PAN ID of any other detectable network.

- **Extended PAN ID:** A 64-bit value called the Extended PAN ID (EPID) is used in forming the network and subsequently modifying the network, if necessary. This identifier can be pre-set to a random value in the user application that runs on the Coordinator. Alternatively, the identifier can be pre-set to zero. In this case, the Coordinator adopts its own 64-bit IEEE/MAC address as the Extended PAN ID when the network starts. This is a sure way of obtaining a globally unique value (see Section 3.2.4).

When a Router or End Device first tries to find a network to join, it uses the Extended PAN ID in either of following ways:

- If an Extended PAN ID has been pre-set in the user application for the Router or End Device, the node joins the network that has this Extended PAN ID (provided this network is detected).
- If there is no pre-set Extended PAN ID for the Router or End Device, the node joins the first network detected, irrespective of the Extended PAN ID. The joining node then learns the Extended PAN ID of its network. It later uses this identifier to rejoin the network if, for some reason, it loses contact with the network (the node is orphaned).

For more information on joining a network, refer to Section 3.3.2.

Note: At the Application level, you only need to be concerned with the Extended PAN ID, as the allocation and use of the PAN ID is transparent to the application.

Parent topic: Network level concepts

Parent topic:*ZigBee PRO architecture and operation*

Network creation This section outlines the process of starting and forming a ZigBee PRO network:

- Section 3.3.1 describes how the Coordinator starts a network.
- Section 3.3.2 describes how a Router or End Device joins a network as part of the network formation process.

Note: The network formation actions described in this section are performed automatically by the ZigBee stack. The actions required at the application level are described later in Section 6.1, "Forming and Joining a Network".

Starting a Network (Coordinator) The Coordinator is responsible for starting a network. It must be the first node to be started and, once powered on, goes through the following network initialization steps:

Set EPID and Coordinator address The Coordinator first sets the Extended PAN ID (EPID) for the network and the device's own network address:

- Sets the EPID to the 64-bit value specified in the Coordinator's application (if this value is zero, the EPID will be set to the 64-bit IEEE/MAC address of the Coordinator device)

- Sets the 16-bit network address of the Coordinator to 0x0000

Parent topic: Starting a Network (Coordinator)

Select radio channel The Coordinator then selects the radio channel in which the network will operate, within the chosen RF band. The Coordinator performs an Energy Detection Scan in which it scans the RF band to find a quiet channel (the scan can be programmed to 'listen' to specific channels). The channel with the least detected activity is chosen.

Parent topic: Starting a Network (Coordinator)

Set the PAN ID of the network Once the radio channel has been selected, the Coordinator chooses a 16-bit PAN ID for the network. To do this, it listens in the channel for traffic from other networks and identifies the PAN IDs of these networks (if any). To avoid conflicts, the Coordinator assigns its own network a random PAN ID that is not in use by another network.

Parent topic: Starting a Network (Coordinator)

Receive join requests from other devices The Coordinator is now ready to receive requests from other devices (Routers and End Devices) to wirelessly connect to the network through it. For more information on joining a network, refer to Section 3.3.2.

Parent topic: Starting a Network (Coordinator)

Parent topic: Network creation

Joining a network (Routers and End Devices) Routers and End Devices can join an existing network already created by a Coordinator. The Coordinator and Routers have the capability to allow other nodes to join the network through them. The join process is as follows:

Search for network The new node first scans the channels of the relevant RF band to find a network. Multiple networks may operate, even in the same channel, and the selection of a network is the responsibility of the application (for example, this decision could be based on a pre-defined Extended PAN ID).

Parent topic: Joining a network (Routers and End Devices)

Select parent The node now selects a parent node within the chosen network by listening to network activity. The node may be able to 'hear' multiple Routers and the Co- ordinator from the network. Given a choice of parents, the node chooses the parent with the smallest depth in the network - that is, the parent closest to the Coordinator (which is at depth zero).

Parent topic: Joining a network (Routers and End Devices)

Request joining The node sends a message to the desired parent, asking to join the network.

Parent topic: Joining a network (Routers and End Devices)

Receive response The node now waits for a response from the potential parent, which determines whether the node is a permitted device and whether the parent is currently allowing devices to join. To determine whether the joining node is a permitted device, the parent consults the Trust Centre (if it is not the Trust Centre itseIf). If these criteria are satisfied, the parent will then allow the node to join the network as its child. In its acceptance response to its new child, the parent will include the 16-bit network address that it has randomly allocated to the child (see Section 3.2.4).

If the potential parent is unable to accept the node as a child, a rejection response is sent. to the node, which must then try another potential parent (or another network).

Parent topic: Joining a network (Routers and End Devices)

Learn network IDs The new node learns the PAN ID and Extended PAN ID of the network, as well as the network address that it has been assigned. It will need the PAN ID for communications with the network and will need the Extended PAN ID if, at some point in the future, it needs to rejoin the network (it will also be able to re- use its network address if it later rejoins the network).

A Router or Coordinator can be configured to have a time-period during which joins are allowed, controlled by its 'permit joining' status. The join period may be initiated by a user action, such as pressing a button. An infinite join period can also be set, so that child nodes can join the parent node at any time.

Note: When an orphaned node attempts to rejoin the network, the 'permit joining' status of a potential parent is ignored. Thus, the node is able to rejoin the network through a parent on which 'permit joining' is disabled.

Parent topic: Joining a network (Routers and End Devices)

Parent topic: Network creation

Parent topic:*ZigBee PRO architecture and operation*

Application level concepts This section describes some key concepts required at the application level.

Multiple applications and endpoints A node may have several applications running on it - for example, a node in a smart home network may incorporate an occupancy sensor and a light switch, each of which is an application. In fact, each application implements a ZigBee device type (see Section 2.10). Access to application instances is provided through endpoints, which act as communication ports for the applications.

In order to direct a message to the appropriate application instance on a node, the relevant endpoint must be specified. Endpoints are numbered from 1 to 240.

Therefore, to communicate with a remote application instance in a ZigBee network, you need to supply the address of the remote node together with the required endpoint number on the node.

Endpoint 255 is the broadcast endpoint number - the same data can be sent to all application instances on a node by sending the message to this endpoint number.

Parent topic: Application level concepts

Descriptors An application may need to obtain information about the nodes of the network in which it runs, as described in Section 3.4.6. For this, it uses information stored in descriptors in the nodes.

There are three mandatory descriptors and two optional descriptors stored in a node. The mandatory descriptors are the Node, Node Power and Simple descriptors, while the optional descriptors are called the Complex and User descriptors

For each node, there is only one Node and Node Power descriptor, but there is a Simple descriptor for each endpoint. There may also be Complex and User descriptors in the device.

The Node, Node Power and Simple descriptors are outlined below. For full details of the descriptors, refer to Section 9.2.1.

Simple descriptor The Simple descriptor for an application includes:

- The endpoint on which the application runs and communicates
- The ZigBee device type that the application implements
- The ZigBee clusters that the device type implements
- Whether there are corresponding Complex and User descriptors
- Lists of input and output clusters (see Section 3.4.1) that the application uses and provides, respectively

Parent topic: Descriptors

Node descriptor The Node descriptor contains information on the capabilities of the node, including:

- Type (End Device, Router or Coordinator)
- Frequency band in use (868 MHz, 902 MHz or 2400 MHz)
- IEEE 802.15.4 MAC capabilities - that is, whether:
  - the device can be a PAN Coordinator
  - the node implements a Full-Function or Reduced-Function IEEE 802.15.4 device
  - the device is mains powered
  - the device is capable of using MAC security
  - the receiver stays on during idle periods
- Manufacturer code
- Stack compliance revision (of the ZigBee PRO Core specification to which the stack complies - prior to Revision 22/ZigBee2017, these bits were reserved and set to zero)
- Maximum buffer size (the largest data packet that can be sent by an application in one operation)

Parent topic: Descriptors

Node power descriptor The Node Power descriptor contains information on how the node is powered:

- Power mode - whether the device receiver is on all the time, or wakes up periodically as determined by the network, or only when an application requires it (for example, during button press).
- Available power sources - indicates whether the mains supply, or rechargeable or disposable batteries (or any combination) can be used to power the device.
- Current power sources - indicates which power source (mains supply, or rechargeable or disposable batteries) is currently being used to power the device.
- Current power source level - indicates the level of charge of the current power source.

Parent topic: Descriptors

Parent topic: Application level concepts

Application profiles One of the aims of ZigBee 3.0 is to unify the market-specific ZigBee application profiles that collect together related device types. Application profile identifiers are still needed in ZigBee 3.0 (this ensures backward compatibility with earlier ZigBee versions) but there has been some consolidation of the identifiers - for example, ZigBee Light Link and Home Automation are both covered by the application profile ID 0x0104, which now corresponds to the ZigBee Lighting and Occupancy (ZLO) devices. Profile matching rules exist and are detailed in the ZigBee 3.0 specification.

Parent topic: Application level concepts

Device types To ensure the interoperability of ZigBee nodes from different manufacturers, the ZigBee Alliance has defined a set of standard device types. A device type (for example, Dimmable Light) is a software entity which defines the functionality of a device. This functionality is itself defined by the clusters included in the device type, where each cluster corresponds to a specific functional aspect (for example, Level Control) of the device. For more information on clusters, refer to Section 3.4.5.

A device is an instance of a device type and is implemented by an application that runs on an endpoint. A device type usually supports both mandatory and optional clusters, so a device can be customized in terms of the optional clusters used. The device type implemented by an application is specified in the application's Simple Descriptor (see Section 3.4.2.1). A node may implement more than one device type, each corresponding to a device application that runs on its own endpoint.

Every ZigBee 3.0 node must employ the ZigBee Base Device, which provides a framework for using ZigBee device types and handles fundamental operations such as commissioning (this device does not need an endpoint).

The NXP implementations of the ZigBee device types and ZigBee Base Device are described in the *ZigBee Devices User Guide (JNUG3131)*.

Parent topic: Application level concepts

Clusters and attributes A data entity (for example, temperature measurement) handled by a ZigBee endpoint is referred to as an attribute. The application may communicate via a set of attributes - for example, a thermostat application may have attributes for temperature, minimum temperature, maximum temperature and tolerance.

ZigBee applications use the concept of a "cluster" for communicating attribute values. A cluster consists of a set of related attributes together with a set of commands to interact with the attributes - for example, commands for reading the attribute values.

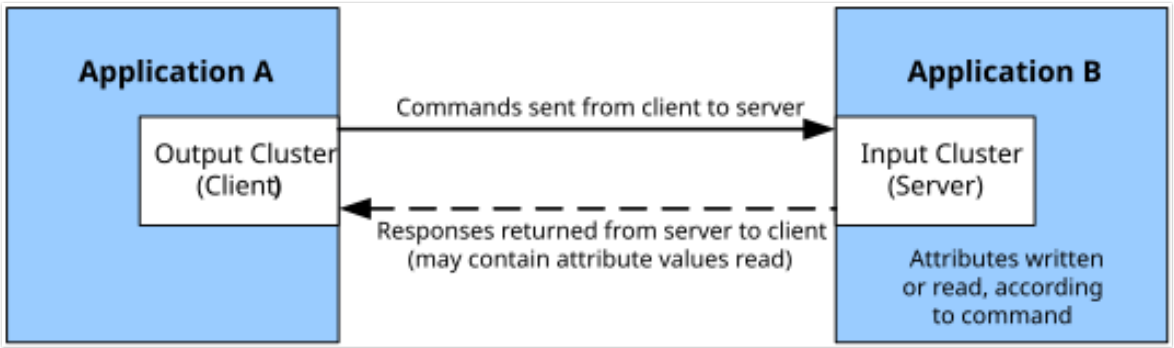
A cluster corresponds to a specific piece of functionality for a device application. The total functionality for the application is determined by the ZigBee device type that it implements and the clusters that the device type uses (see Section 3.4.4). Thus, clusters are the functional building blocks of devices.

A cluster has two aspects, which are respectively concerned with receiving and sending commands (one or both aspects may be used by a ZigBee application):

- **Input Cluster or Server Cluster:** This side of a cluster is used to store attributes and receive commands to manipulate the stored attributes (to which the cluster may return responses) - for example, an input cluster would store a temperature measurement and associated attributes, and respond to commands which request readings of these attributes.
- **Output Cluster or Client Cluster:** This side of a cluster is used to manipulate attributes in the corresponding input cluster by sending commands to it (and receiving the responses). Normally, these are write commands to set attribute values and read commands to obtain attribute values (the read values being returned in responses).

The Output/Client and Input/Server sides of a cluster are illustrated in below figure.

Input (Server) and Output (Client) Clusters



The input clusters and output clusters communicated via an endpoint are listed (separately) in the endpoint's Simple descriptor (see Section 3.4.2.1).

For consistency and interoperability, the ZigBee Alliance have defined a number of standard clusters for different functional areas. These are collected together in the ZigBee Cluster Library (ZCL). Thus, developers can use standard clusters from the ZCL in their device applications. The ZCL is fully detailed in the *ZigBee Cluster Library Specification (075123)* from the ZigBee Alliance. The NXP implementation of these clusters is detailed in the *ZigBee Cluster Library User Guide (JNUG3132)*.

A Default cluster (with ID of 0xFFFF) is also available. If the Default cluster is present on an endpoint and a message is received which is destined for a cluster that is not in the endpoint's list of supported input clusters, this message will still be passed to the application (provided it comes from a defined application profile). If it is required, the Default cluster must be explicitly added to the endpoint (see Section 13.4.3).

Parent topic: Application level concepts

Discovery The ZigBee specification provides the facility for devices to find out about the capabilities of other nodes on a network, such as their addresses, which types of applications are running on them, their power source and sleep behavior. This information is stored in descriptors (see Section 3.4.6) on each node, and is used by the enquiring node to adapt its behavior to the requirements of the network.

Discovery is typically used when a node is being introduced into a user-configured network, such as a domestic security or lighting control system. It may require the user to press a button or similar to begin the process of integration of the device into the network. The first task is to find out if there are any appropriate devices with which the new node can communicate.

Device discovery Device discovery returns information about the addresses of a network node. The retrieved information can be the IEEE/MAC address of the node with a given network address, or the network address of a node with a given IEEE/MAC address. If the node being interrogated is a Router or Coordinator, it may optionally supply the addresses of all the devices that are associated with it, as well as its own address. In this way, it is possible to discover all the devices on a network by requesting this information from the Coordinator (network address 0x0000) and then using the list of addresses corresponding to the children of the Coordinator to launch other queries about their child nodes.

Parent topic: Discovery

Service discovery Service discovery allows a node to request information from a remote node about the remote node's capabilities. This information is stored in a number of descriptors (see Section 3.4.2) on the remote node. It includes the following:

- The device type and capabilities of the node.
- The power characteristics of the node.
- Information about each application running on the node.
- Optional information such as serial numbers.
- Other user-defined information - for example, easily understandable names such as '*MtgRoomLight*'.

Requests for these descriptors are made by a device during the discovery process that is typically part of the device's configuration and integration into a ZigBee network.

Parent topic: Discovery

Parent topic: Application level concepts

ZigBee Device Objects (ZDO) A special application, common to all ZigBee devices, is provided to manage the various processes that have been described. This application is the ZigBee Device Objects or ZDO. It resides in the Application layer of a node, and can communicate with remote nodes via endpoint 0 using the ZigBee Device Profile (ZDP) and associated clusters. It has the following roles:

- Defines the type of network device: Coordinator, Router or End Device
- initializes the node to allow applications to be run
- Performs the device discovery and service discovery processes
- Implements the processes needed to allow a Coordinator to create a network, and Routers and End Devices to join and leave a network
- Initiates and responds to binding requests (see Section 3.6.2)
- Provides security services which allow secure relationships to be established between applications
- Allows remote nodes to retrieve information from the node, such as Routing and Binding tables, and to perform remote management of the node, such as instructing it to leave the network

The ZDO uses services within the stack to implement these roles and provides a means of allowing user applications to access stack services.

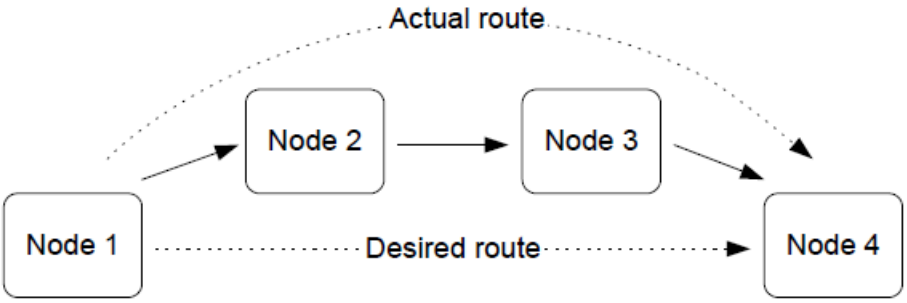
Parent topic: Application level concepts

Parent topic:*ZigBee PRO architecture and operation*

Network routing The basic operation of a network is to transfer data from one node to another. The data is sourced from an input (possibly a switch or a sensor) on the originating node, and is communicated to another node which can interpret and use the data.

In the simplest data communication, the data is transmitted directly from the source node to the destination node. However, if the two nodes are far apart or in a difficult environment, direct communication may not be possible. In this case, it is necessary to send the data to another node within radio range, which then passes it on to another node, and so on until the desired destination node is reached - that is, to use one or more intermediate nodes as stepping stones. The process of receiving data destined for another node and passing it on is known as routing.

Message routing



Routing allows the range of a network to be extended beyond the distances supported by direct radio communication. Remote devices can join the network by connecting to a Router.

Note: Application programs in intermediate nodes are not aware of the relayed message or its contents - the relaying mechanism is handled by the ZigBee stack.

Message addressing and propagation If a message sent from one node to another needs to pass through one or more intermediate nodes to reach its final destination (up to 30 such hops are allowed), the message carries two destination addresses:

- Address of the final destination.
- Address of the node which is the next "hop".

ZigBee PRO is designed for Mesh networks (see Section 3.2.2) in which the message propagation path (the route) depends on whether the target node is in radio range:

- If the target node is in range, only the "final destination" address is used.
- If the target node is not in range, the "next hop" address is that of the first node in the route to the final destination.

The "next hop" address is determined using information stored in a Routing table on the routing node (Router or Coordinator). An entry of this table contains information for a remote node, including the network addresses of the remote node and of the next routing node in the route to the remote node. Thus, when a message is received by a routing node, it looks for the destination address in its Routing table and extracts "next hop" address from this table to insert into the message. The message is then passed on and propagation continues in this way until the target node is reached.

Note: If the message originates from an End Device, the message is always first passed to the source node's parent before being passed on.

Parent topic: Network routing

Route discovery The ZigBee stack network layer supports a 'route discovery' facility which finds the best available route to the destination, when sending a message. A message is normally routed along an already discovered mesh route, if one exists. Otherwise, the routing node (Router or the Coordinator) involved in sending the message initiates a route discovery. Once complete, the message is sent along the calculated route.

The mechanism for route discovery between two End Devices has the following steps:

1. A route discovery broadcast is sent by the parent of the source End Device, and contains the destination End Device's network address.
2. All routing nodes eventually receive the broadcast, one of which is the parent of the destination End Device.
3. The parent of the destination node sends back a reply addressed to the parent of the source node.

4. As the reply travels back through the network, the hop count and a signal quality measure for each hop are recorded. Each routing node in the path can build a Routing table entry containing the best path to the destination End Device.

The best path is usually the one with the least number of hops. However, if a hop on the most direct route has a poor signal quality, a greater chance that retries would be needed. In such cases, a route with more hops might be chosen.

1. Eventually each routing node in the path has a Routing table entry and the route from source to destination End Device is established. Note that the corresponding route from destination to source is not known - the route discovered is unidirectional.

A source Router implements route discovery in a similar way to the above except the Router broadcasts its own route discovery message (without needing its parent to do this). Similarly, the Coordinator broadcasts its own route discovery messages.

Note:

Message routing is performed automatically by the ZigBee stack and is transparent to the user application. If required, route discovery is also automatic and transparent to the application.

Parent topic: Network routing

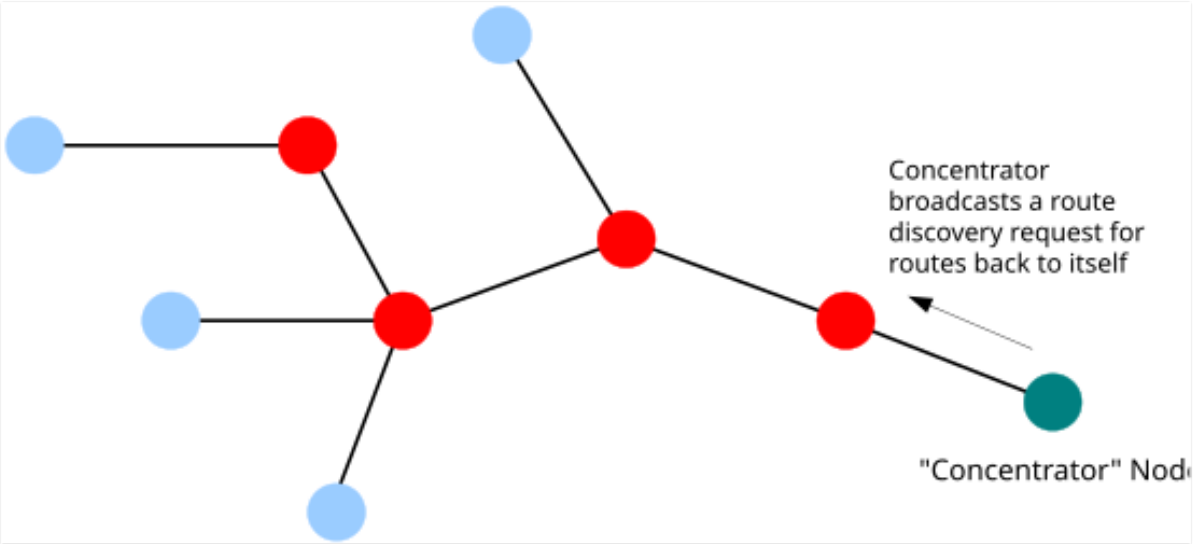
'Many-to-one' routing A common scenario in a wireless network is the need for most network nodes to communicate with a single node that performs some centralized function, for example, a gateway. This node is often referred to as a concentrator.

In order to establish communication with the concentrator, each remote node may initiate a 'route discovery', resulting in a corresponding entry in the Routing table of each routing node along the way. If most network nodes need to communicate with the concentrator, many such route discoveries may be initiated. Where the resulting routes have a common leg, the relevant Routing table entries will not be duplicated but shared. However, a large number of simultaneous route discoveries may require significant memory space in the nodes near the concentrator for the temporary storage of route discovery information, and possibly result in memory overflow and traffic congestion.

A more efficient method of establishing routes to a concentrator is for the concentrator to initiate a 'many-to-one' route discovery for routes from all other network nodes to itself. To do this, the concentrator broadcasts a route discovery request and the Routing tables are updated as the broadcast propagates through the network. Since no responses are generated, the temporary storage of route discovery information is not required and network traffic congestion is minimized.

Many-to-one route discovery is illustrated in the figure below.

'Many to one' routing



In order to avoid the storage of return routes (from the concentrator) in the Routing tables of intermediate nodes, the technique of source routing is used - the outward route taken by a message to the concentrator is remembered by the concentrator and embedded in the response message. In this case, the response message must carry up to 30 addresses of the nodes along the return route (maximum number of hops allowed is 30).

Parent topic: Network routing

Parent topic:*ZigBee PRO architecture and operation*

Network communications This section considers the processes that are needed to allow a network of devices to exchange information and perform useful functions. In order to communicate with each other, two nodes must be compatible in that one node can produce data which the other node can accept and interpret in a meaningful way. For example, a temperature sensor node produces a temperature measurement that a heating controller node can use to control a central heating system.

When a new node joins a network, it must find compatible nodes with which it is able to communicate - this process is facilitated by the Service Discovery mechanism. It must then choose which of the compatible nodes it will communicate with. A method of pairing nodes for easy communication is provided by the binding mechanism.

Note: While you should always use Service Discovery to find compatible nodes, binding is an optional method for pairing compatible nodes.

Service Discovery and binding are covered in the sub-sections below.

Service discovery A device joining a network must be able to find other devices in the network that can use the information it provides, or that can generate the information needed by the device to perform its own function. A node can use Service Discovery to find nodes with which it can communicate. Service Discovery is introduced in Section 3.4.6.

The node requests the required services from other nodes by means of a broadcast message that propagates throughout the network. Any node that has the requested services then unicasts a response back to the requesting node. This means that the requesting node may receive more than one response.

A response includes the network address of the remote node that contains the requested services. The node stores this address locally and the application can then use the address for all future communications to the remote node. This is referred to as direct addressing.

Alternatively, rather than using direct addressing in their communications, two nodes can communicate through the binding mechanism, described in Section 3.6.2 below.

Parent topic: Network communications

Binding Once two nodes have been found to be compatible through Service Discovery (see Section 3.6.1), they may be paired for communication purposes. For example, a light- switch may be paired with a particular light, and we must ensure that this light-switch only ever switches the light that it is intended to control. An easy way to pair nodes for communication is provided by the binding mechanism.

Binding allows nodes to be paired in such a way that a certain type of output data from one node is automatically routed to the paired node, without the need to specify the destination address and endpoint every time. The two nodes must first be bound together using the address and relevant endpoint number for each node - these can be obtained through Service Discovery, described in Section 3.6.1. A binding has a source node and a destination node, relating to the direction in which data is sent between the nodes (from source to destination). The details of a binding are stored as an entry in a binding table, normally held on the source node of the binding or sometimes on another nominated node.

In order to establish a binding, it must be requested in either of the following ways:

- Binding request is submitted to the source node for the binding by either the source node itself or a remote node (not one of the nodes to be bound).
- Binding requests are submitted to the Coordinator by the source and destination nodes for the binding (for example, by pressing a button on each node to generate a binding request). The two binding requests must be received within a certain timeout period.

During the binding process, the Binding table for the source node is updated or, if necessary, created.

Binding occurs at the application level using clusters (described in Section 3.4.5). In order for two applications to be bound, they must support the same cluster.

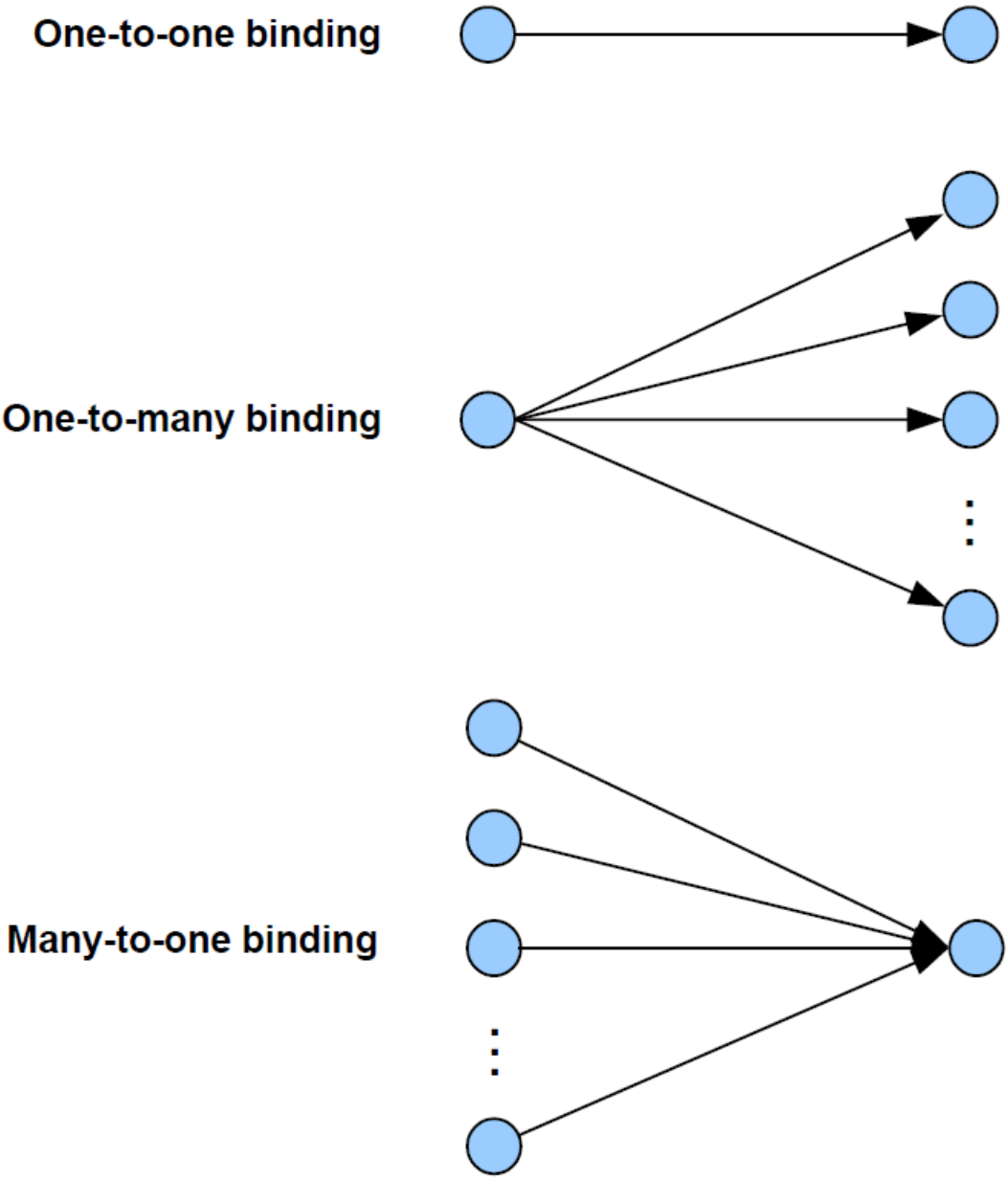
The binding between two applications is specified by:

- The node address and endpoint number of the source of the binding (for example, a light-switch).
- The node address and endpoint number of the destination of the binding (for example, the load controller for a light).
- The cluster ID for the binding.

The following types of binding can be achieved:

- One-to-one: This is a simple binding in which an endpoint is bound to one (and only one) other endpoint, requiring a single Binding table entry.
- One-to-many: This is a binding in which a source endpoint is bound to more than one destination endpoint. The binding is achieved by having multiple Binding table entries for the same source endpoint.
- Many-to-one: This is a binding in which more than one source endpoint is bound to a single destination endpoint. The binding is achieved by multiple nodes having one-to-one bindings for the same destination endpoint.

These are illustrated in the figure below.



As an example of these bindings, consider a switch and load controller for lighting:

- In the one-to-one case, a single switch controls a single light
- In the one-to-many case, a single switch controls several lights
- In the many-to-one case, several switches control a single light, such as a light on a staircase, where there are switches at the top and bottom of the stairs, either of which can be used to switch on the light

It is also possible to envisage many-to-many bindings where in the last scenario there are several lights on the staircase, all of which are controlled by either switch.

The way bindings are configured depends on the type of network (described in Section2.6), as follows:

- Pre-configured system: Bindings are factory-configured and stored in the application image.
- Self-configuring system: Bindings are automatically created during network installation using discovery software that finds compatible nodes/clusters.

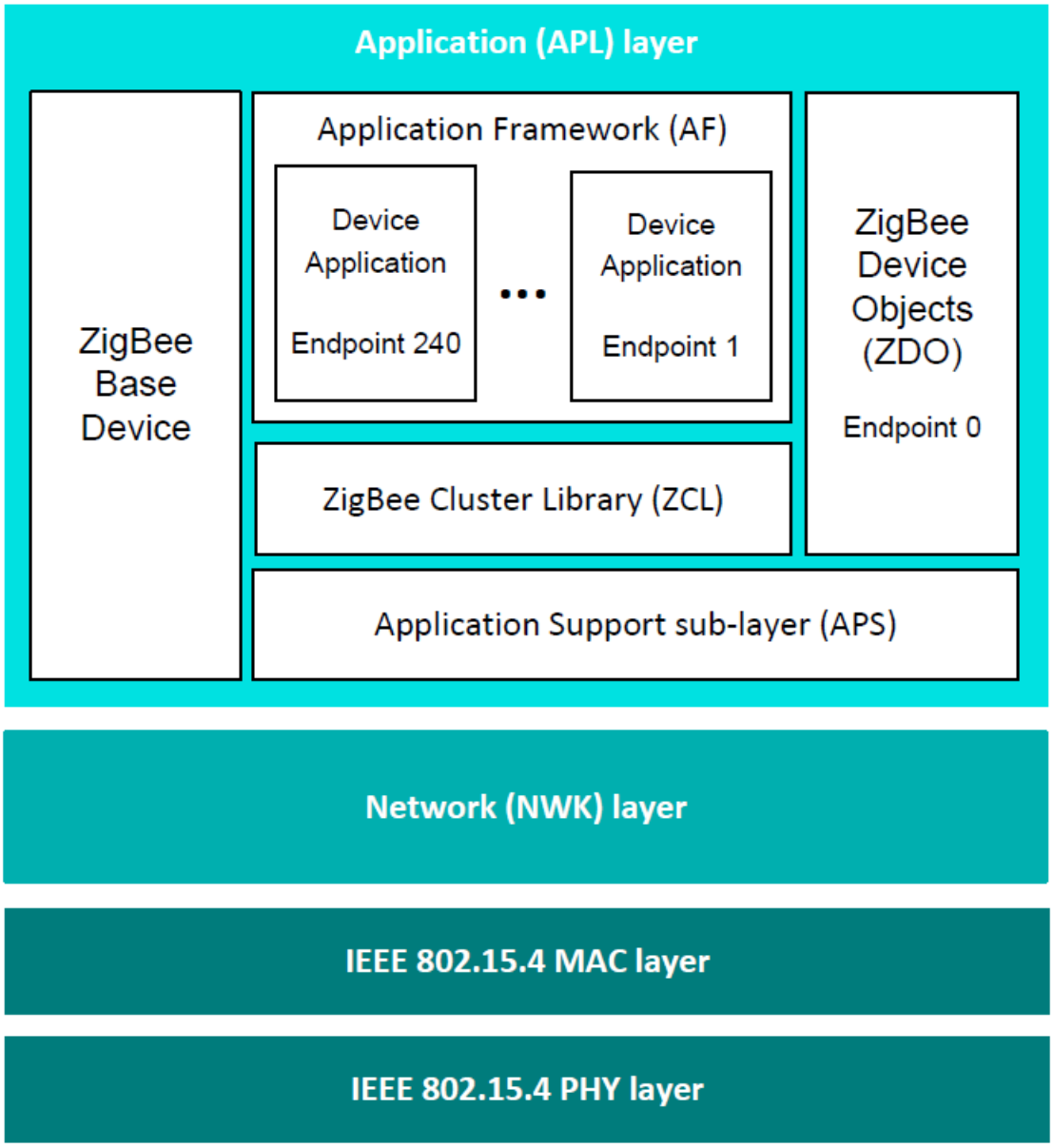
- Custom system: Bindings are created manually by the system integrator or installation technician, who may use a graphical software tool to draw binding lines between clusters on nodes.

Parent topic: Network communications

Parent topic:*ZigBee PRO architecture and operation*

Detailed architecture This section elaborates on the simplified software architecture presented in "Section 3.1". The detailed architecture is illustrated in the figure below.

Detailed software architecture



Software levels The preceding figure Figure shows the architecture diagram (from top to bottom).

Application (APL) Layer This includes:

- Applications: Up to 240 application instances can be supported on a single ZigBee node. Each application instance communicates via an endpoint, where endpoints are numbered between 1 and 240 (note that endpoint 0 is reserved for the ZDO of the node - see below).
- Application Framework (AF): The AF facilitates interaction between the applications and the APS layer (see below) through an interface known as a Service Access Point or SAP.
- ZigBee Device Objects (ZDO): The ZDO represents the ZigBee node type of the device (Coordinator, Router, or End Device) and has a number of communication roles. The ZDO communicates via endpoint 0. For more information, refer to "Section 3.4.7".
- ZigBee Base Device: This device is required for all ZigBee 3.0 nodes and deals with essential tasks for the whole node, such as commissioning. It does not occupy an endpoint.
- ZigBee Cluster Library (ZCL): The ZCL provides the standard ZigBee clusters used by the device applications that run on the endpoints.
- Application Support sub-layer (APS): The APS layer is responsible for:
  - Communicating with the relevant application - for example, when a message arrives to illuminate an LED, the APS layer relays this instruction to the responsible application using the endpoint information in the message.
  - Maintaining binding tables (see "Section 3.6.2") and sending messages between bound nodes.
  - Providing communication with the Trust Centre to obtain authorization.

The APS layer has an associated database, called the APS Information Base (AIB). This contains attributes that mainly relate to system security.

Parent topic: Software levels

Network (NWK) layer The NWK layer handles network addressing and routing by invoking actions in the MAC layer. It provides services for:

- Starting the network
- Assigning network addresses
- Adding devices to and removing them from the network
- Routing messages to their intended destinations
- Applying security to outgoing messages
- Implementing route discovery and storing Routing table information

The NWK layer has an associated database, called the NWK Information Base (NIB). This contains attributes required in the management of the NWK layer.

Parent topic: Software levels

Physical/Data link layers This consists of the IEEE 802.15.4 PHY and MAC layers, described in Section 3.1,"Architectural overview".

Note: The Security Service Provider (not shown in the figure) spans the APS and NWK layers, providing security services - for example, security key management, datastream encryption and decryption. It may use hardware functions provided in the node to perform the encode and decode operations efficiently.

Parent topic: Software levels

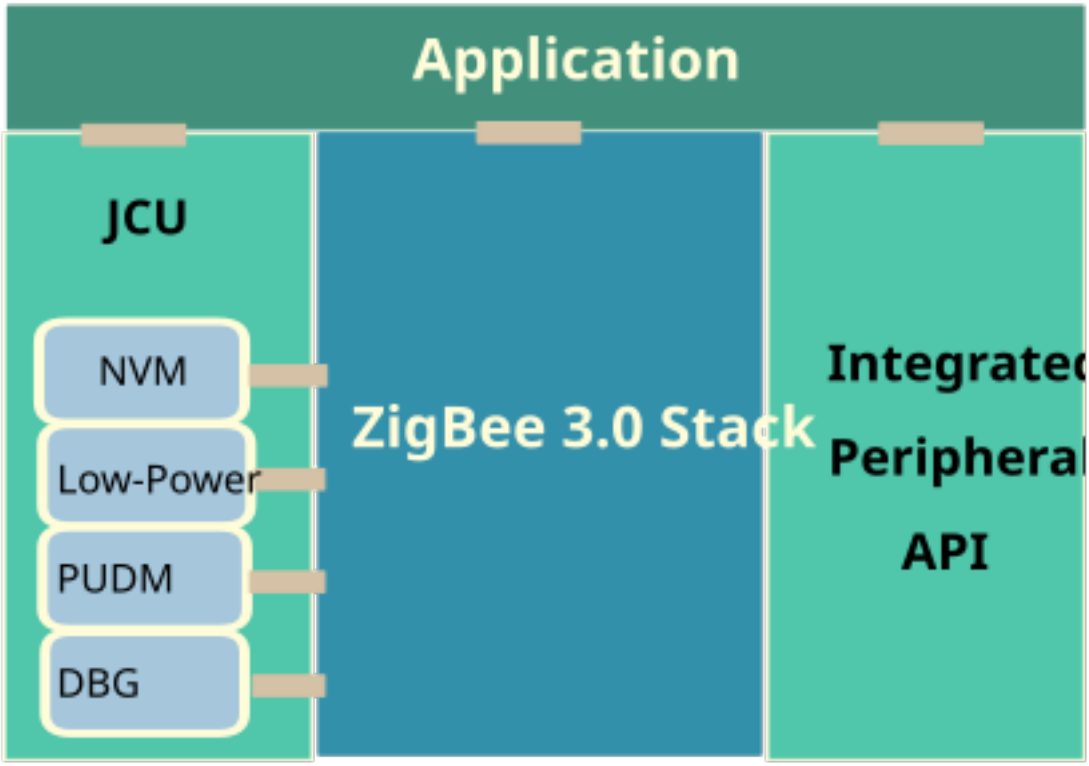
Parent topic: Detailed architecture

Parent topic:*ZigBee PRO architecture and operation*

ZigBee Stack Software This chapter introduces the NXP ZigBee 3.0 stack software.

Software overview The NXP ZigBee 3.0 software provides all components of the ZigBee stack detailed in Section 2.7, Detailed architecture. In addition, it includes the JN51xx Core Utilities (JCU). The basic architecture of this software, in relation to the wireless network application, is illustrated in the figure below.

Overview of NXP ZigBee software architecture



The NXP ZigBee 3.0 software includes Application Programming Interfaces (APIs) to facilitate simplified application development for wireless networks. These APIs consist of C functions that can be incorporated directly in application code.

Two general categories of API are supplied:

- ZigBee PRO APIs - see ZigBee PRO APIs
- JCU APIs - see JCU APIs

The above figure also shows the Integrated Peripherals API that can be used to interact with the on-chip hardware peripherals of the device. This API is described in the MCUXpresso SDK API Reference Manual ((*MCUXSDKJN5189APIRM* or *MCUXSDKK32W041APIRM*).

In addition, the ZigBee Cluster Library (ZCL) provides APIs for the individual clusters, as well as more general ZCL functions. The ZCL is located within the stack block.

All the above APIs are supplied in the ZigBee 3.0 Software Developer's Kit (SDK). For more details on the SDK, refer to Section 5.1, Development environment and resources).

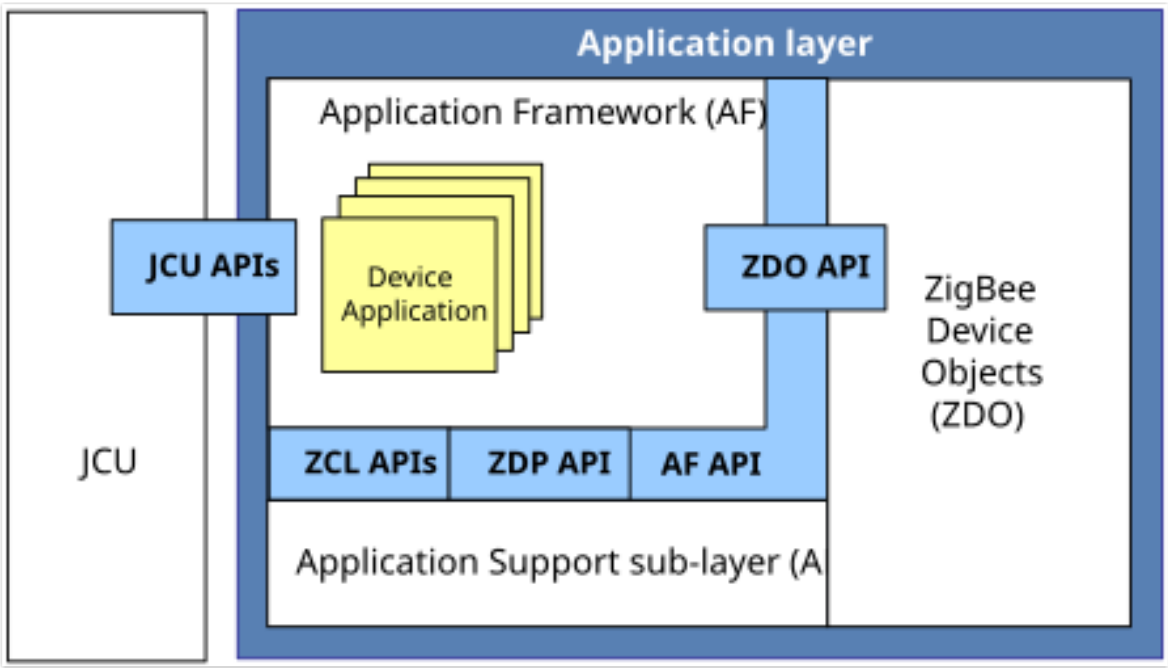
ZigBee PRO APIs The ZigBee PRO APIs are concerned with network-specific operations and easy interaction with the ZigBee PRO stack from the application code. These C-function APIs are supplied in the ZigBee 3.0 SDK (see Section 5.1, Development environment and resources).

There are three ZigBee PRO APIs:

- **ZigBee Device Objects (ZDO) API:** Concerned with the management of the local device (for example, introducing the device into a network)
- **ZigBee Device Profile (ZDP) API:** Concerned with the management of remote devices (for example, device discovery, service discovery, binding)
- **Application Framework (AF) API:** Concerned with creating data frames for transmission and modifying device descriptors

The locations of these APIs, as well as the JCU and ZCL APIs, are illustrated in the figure below.

Locations of ZigBee 3.0 APIs



Note: The C functions of all the ZigBee PRO APIs are fully detailed in Part II: Reference Information of this manual.

Parent topic: Software overview

JCU APIs The Core Utilities (JCU) provide an easy-to-use interface to simplify the programming of a range of non-network-specific operations. These utilities/modules each have a C function API, which allows a module to be used from a user application. The JCU is supplied in the ZigBee 3.0 SDK.

The JCU modules are outlined below:

- **Non-Volatile Memory Manager (NVM):** This module handles the storage of context and application data in Non-Volatile Memory (NVM), and the retrieval of this data. It provides a mechanism by which the device can resume operation without loss of continuity following a power loss.
- **Low-Power:** This module manages the transitions of the device into and out of low-power modes, such as sleep mode.
- **Protocol Data Unit Manager (PDUM):** This module is concerned with managing memory, as well as inserting data into messages to be transmitted and extracting data from messages that have been received.

Note: The JCU modules are fully described in the *DK6 Core Utilities User Guide (JNUG3133)*.

Parent topic: Software overview

Parent topic:*ZigBee Stack Software*

Summary of API functionality This section summarizes the roles of the NXP ZigBee PRO and JCU APIs in an application. The table below indicates the APIs needed for the different functionality that might be required in your code:

Use of ZigBee PRO and JCU APIs

Functionality	ZigBee PRO APIs	JCU APIs
Essential functionality, including network formation and management	**ZDO API:** Network formation and local network management **ZDP API:** Network discovery and remote network management	-
Basic data transfer	**AF API:** Sending and receiving data messages	**PDUM API:** Assembling and disassembling data messages
Binding endpoints for data transfers between them	**ZDO API:** Basic binding **ZDP API:** Manipulation of remote Binding tables	-
Low-power modes (Sleep and Doze)	-	**Low-Power API:** Managing low-power modes
Preserving context data (for example, for resuming operation after sleep without memory held)	-	**NVM API:** Saving and restoring context data
Network security	**ZDO API:** Managing security	-

Important points to note:

- ZigBee PRO API function names are prefixed with 'ZPS' (for 'ZigBee PRO Stack' function). The function names also incorporate 'Apl' (for 'Application' function) and the acronym for the API to which the function belongs:
  - ZDO function names include 'Zdo' (for example, **ZPS_eAplZdoPoll()**).
  - ZDP function names include 'Zdp' (for example, **eAplZdpActiveEpRequest()**).
  - AF function names include 'Af' (for example, **ZPS_eAplAfUnicastDataReq()**).
- JCU API function names are prefixed with the acronym for the JCU module to which the function belongs:
  - 'NVM' for NVM functions.
  - 'PWR' for Low-Power functions.
  - 'PDUM' for PDUM functions.

A similar naming convention is used in structures and enumerations.

Parent topic:*ZigBee Stack Software*

Application development overview This chapter provides an overview of the main phases in developing a ZigBee 3.0 wireless network product. It is important that you refer to this chapter, particularly Section 5.3, Development phases, before and during your product development.

You should develop an application program for each node type in your product - Coordinator, Router, and End Device. If a node type has variants, you might need to develop a separate application for each variant. For example, an End Device, which is a Light Sensor and an End Device, which is an On/Off Light Switch in a lighting system.

Development environment and resources This User Guide supports the NXP ZigBee 3.0 Software Developer's Kits (SDKs) for the JN518x and K32W041/K32W061/K32W1/MCXW71/MCXW72/RW612 devices.

Development platform

MCUXpresso NXP MCUXpresso provides an Eclipse-based platform for developing applications for the JN518x and K32W041/K32W061/K32W1/MCXW71/MCXW72 devices. This platform can be obtained from https://community.nxp.com/community/mcuxpresso/mcuxpresso-ide and must be a registered edition. **The required version of MCUXpresso is indicated in the Release Notes for the ZigBee 3.0 SDK.**

For installation and operational instructions, first refer to the *MCUXpresso Installation and User Guide*. More detailed operational instructions are provided in the *MCUXpresso User Guide*, available from the above website.

Attention: Zigbee applications on RW612 platform do not support MCUXPresso IDE projects. Instead they are built with the west tool, refer to the https://github.com/nxp-mcuxpresso/mcux-sdk for more information.

Parent topic: Development platform

Parent topic: Development environment and resources

ZigBee 3.0 SDK The ZigBee 3.0 SDK provides the stack and API software resources needed to develop ZigBee 3.0 applications for the JN518x and K32W041/K32W061/K32W1/MCXW71/MCXW72/RW612 devices and includes:

- ZigBee PRO and IEEE 802.15.4 stack software
- ZigBee PRO APIs
- ZigBee Base Device Behavior (BDB) APIs
- ZigBee Cluster Library (ZCL) APIs
- Connectivity Framework APIs
- ZPS Configuration Editor
- Integrated Peripherals APIs and Board APIs

NXP-specific tools have been devised for MCUXpresso, including the ZPS Configuration Editor, which is provided as an Eclipse plug-in. This tool is used to set network parameters and is described in Chapter 13, *ZPS Configuration Editor*.

This ZigBee 3.0 SDK contains device-specific plug-ins for the MCUXpresso platform.

MCUXpresso must be installed before the ZigBee 3.0 SDK. Refer to Section 5.1.1, Development platform for information on this toolchain.

Parent topic: Development environment and resources

Parent topic:*Application development overview*

Zigbee application support resources While developing your ZigBee 3.0 application for a JN518x or K32W041/K32W061/K32W1/MCXW71/MCXW72/RW612 device, refer to the below User Guides along with JNUG3130:

- *ZigBee Devices User Guide (JNUG3131)*
- *ZigBee Cluster Library User Guide (JNUG3132)*
- *Connectivity Framework Reference Manual*

Refer to the following NXP Application Notes for further support in the development of ZigBee 3.0 applications for the JN518x or K32W041/K32W061/K32W1/MCXW71/MCXW72 devices:

- *ZigBee 3.0 Base Device Template (JN-AN-1217)*
- *ZigBee 3.0 Light Bulbs (JN-AN-1244)*
- *ZigBee 3.0 Controller and Switch (JN-AN-1245)*
- *ZigBee 3.0 Sensors (JN-AN-1246)*
- *ZigBee 3.0 IoT Control Bridge (JN-AN-1247)*

For RW612 devices, refer to *ZigBee 3.0 Light Bulbs**JN-AN-1244* .

Note: The relevant software and documentation resources can be obtained via the Wireless Connectivity area of the NXP website (for the web address, see Support resources.

Parent topic:*Application development overview*

Development phases The main phases of development of a ZigBee 3.0 application are as follows and are conducted in MCUXpresso:

1. **Network Configuration:** Configure the network parameters for the nodes using the ZPS Configuration Editor (refer to Chapter 12, *ZigBee network parameters* and Chapter 13, *ZPS Configuration Editor*).
2. **Application Code Development:** Develop the application code for your nodes using the ZigBee PRO, ZCL, BDB, and JCU APIs.
3. **Application Build:** Build the application binaries for your nodes.
4. **Node Programming:** Load the application binaries into Flash memory on your nodes.

Note: As a starting point for your application development, you may wish to use one or more of the Application Notes listed in Section 5.2, “Zigbee application support resources”.

Parent topic:*Application development overview*

Application coding with ZigBee PRO APIs This chapter outlines how to use functions of the NXP ZigBee PRO APIs to perform common operations required in a ZigBee PRO wireless network application.

The operations covered in this chapter are divided into the following areas:

- Forming a ZigBee PRO wireless network (Section 6.1)
- Discovering the properties of the formed network (Section 6.2)
- Managing group addresses (Section 6.3)
- Binding nodes for easy communication between them (Section 6.4)
- Transferring data between nodes (Section 6.5)
- Leaving and rejoining the network (Section 6.6)
- Return codes and extended error handling (Section 6.7)
- Implementing ZigBee security (Section 6.8)
- Using support software features - message queues and timers (Section 6.9)
- Using advanced features (Section 6.10)

Many of the functions referenced in this chapter are non-blocking functions that submit a request to the relevant node(s) of the network and then return - these functions have **Request** or **Req** in their names. The recipient of the request normally replies by sending a response to the

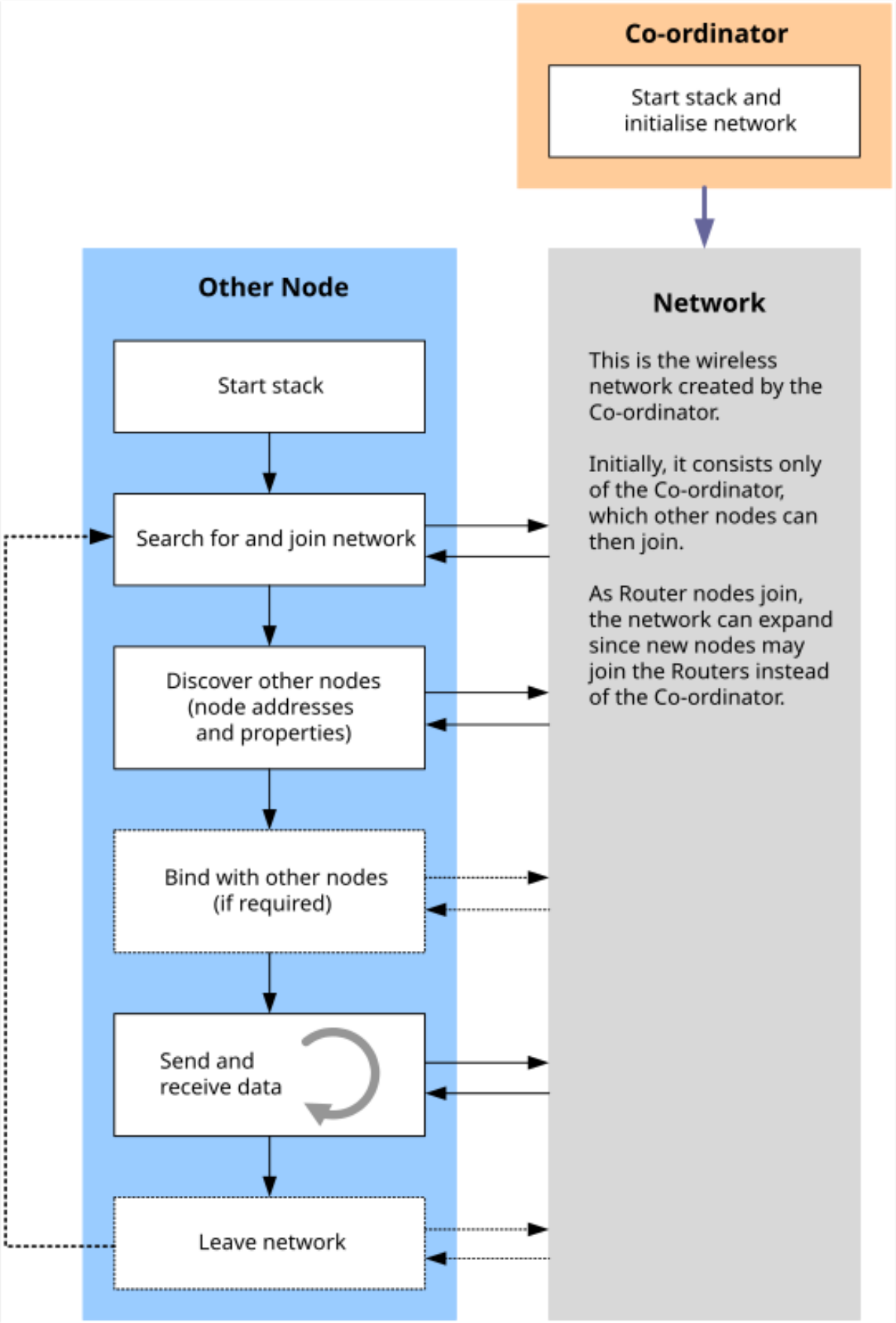
node that initiated the request. Once received, this response message can be collected using the function **ZQ_bZQueueReceive()** - see General queue management.

The ZigBee PRO API functions mentioned in this chapter are fully detailed in *Part II Reference Information (chapter 7 to chapter 12)*. See Organization of this manual.

Note: Further assistance in developing your own ZigBee 3.0 applications is provided in a range of NXP Application Notes (see Section 5.2, Zigbee application support resources).

The main stages of the life-cycle of a wireless network are illustrated in the figure below. These stages incorporate many of the high-level operations described in this chapter.

Wireless Network Life-cycle


Co-ordinator
Start stack and initialise network
Other Node
Start stack
Search for and join network
Discover other nodes (node addresses and properties)
Bind with other nodes (if required)
Send and receive data
Leave network
Network
This is the wireless network created by the Co-ordinator.
Initially, it consists only of the Co-ordinator, which other nodes can then join.
As Router nodes join, the network can expand since new nodes may join the Routers instead of the Co-ordinator.

Forming and joining a network This section describes how to form a wireless network by first starting the Coordinator and then starting the other nodes, which join the network initiated by the Coordinator.

Note: In order to start any network node, certain configuration values must have been pre-set for the

application. This configuration is performed using the steps described in *Chapter 13, ZPS Configuration Editor*.

At initialization, the same function calls are needed for all node types. However, once started, the stack performs initialization tasks according to the specific node type, as described in Section 6.1.1 and Section 6.1.2. These function calls are listed below, in the required order:

1. **PDUM_vInit()** to initialize the JCU Protocol Data Unit Manager (PDUM).
2. **PWR_Init()** to initialize the Low-Power module in order to facilitate low-power modes such as sleep and doze.
3. **NvModuleInit()** to initialize the JCU Non-Volatile Memory Manager (NVM) in order to save context and application data for retrieval after a power break.
4. **eZCL_initialise()** to initialize the ZigBee Cluster Library (ZCL).
5. **eZCL_Register()** for a custom device type, or the equivalent registration function for a standard ZigBee device type, to register an endpoint for the application.
6. **zps_eAplAfInit()** to initialize the Application Framework.
7. **BDB_vInit()** to initialize the ZigBee Base Device.
8. **zps_eAplZdoStartStack()** to start the ZigBee PRO stack.

Note:

- The ZigBee PRO stack can later be reset to its default state (deleting context data except NWK frame counters) using the zps_vDefaultStack() function.
- The IEEE 802.15.4 MAC capabilities of a Router or End Device can be configured by the application using zps_vAplAfSetMacCapability() function.

Starting the Coordinator The Coordinator must be the first node to be started. This node is pre-configured using the ZPS Configuration Editor. The functions that must be called in the Coordinator application to initialize the node are those listed at the start of this (Section 6.1).

Once the stack has been started using **zps_eAplZdoStartStack()**, the Coordinator works through the following process to establish a network:

Setting the radio channel for the network The choice of 2.4-GHz band channel for the network is pre-configured via the the ZPS Configuration Editor (see Section 13.4.3). It is either a fixed channel in the range 11-26 or a set of channels from which the best channel is selected by the Coordinator. In the latter case, the Coordinator performs an energy scan of the possible channels and chooses the quietest channel.

Parent topic: Starting the Coordinator

Setting the Extended PAN ID for the network The 64-bit Extended PAN ID (EPID) for the network is obtained as follows:

- A pre-configured value may be set in the advanced device parameter *APS Use Extended PAN ID*in the ZPS Configuration Editor (see Section 13.4.4).
- If the pre-set value is zero, the Coordinator uses its own IEEE/MAC address as the EPID.

Note: The application might override the EPID value set by the ZPS Configuration Editor by calling **zps_eAplAibSetApsUseExtendedPanId()** before calling **zps_eAplZdoStartStack()**.

Parent topic: Starting the Coordinator

Accepting join requests from other devices (if enabled) The Coordinator may now allow other devices (Routers and End Devices) to join the network as its children, enabling the network to grow. A maximum number of (direct) children of the Coordinator is pre-set via the advanced network parameters *Active Neighbor Table Size*and *Child Table Size*in the ZPS Configuration Editor (see Section 13.4.4, Setting advanced device parameters), beyond which the Coordinator does not accept any further join requests from prospective children.

Note:

The initial 'permit joining' status is pre-set via the Coordinator parameter Permit Joining Time in the ZPS Configuration Editor.

If this is initially disabled, the Coordinator may not accept children until joining has been enabled using **zps_eAplZdoPermitJoining()**.

However, the 'permit joining' status is ignored during a join in which the pre-set EPID on the joining device is non-zero and during any rejoin (see Section 6.6.2). The above function can be used at any time to allow joinings for a limited time-period or indefinitely, and can also be used to disable joinings.

Once the Coordinator (and therefore network) has started, the stack event zps_EVENT_NWK_STARTED is generated on the device. If the Coordinator fails to start, the stack event zps_EVENT_NWK_FAILED_TO_START is generated.

When a node joins the Coordinator, the stack event zps_EVENT_NWK_NEW_NODE_HAS_JOINED is generated on the Coordinator.

Parent topic: Starting the Coordinator

Parent topic: Forming and joining a network

Starting Routers and End Devices A Router or End Device is pre-configured using the ZPS Configuration Editor. The functions that must be called in a Router or End Device application to initialize the node are listed at the beginning of this section (Section 6.1, Forming and joining a network).

Note: The start-up and join process described in this section is for a first-time join (cold start) only and not for a rejoin (which is described in Section 6.6.2).

Once the stack has been started using **zps_eAplZdoStartStack()**, a Router or End Device works through the following process to join a network:

1. Searches for a network to join
2. Selects a network to join
3. Submits a join request to network
4. Records the network's EPID for application use
5. Router accepts join requests from other devices (if enabled)

These processes are described in detail in the following sections.

Searches for a network to join As part of the **zps_eAplZdoStartStack()** function call, the device searches for networks by listening for beacons from Routers and Coordinators of ZigBee PRO networks in the neighborhood. The radio channel for this search is pre-configured via the ZPS Configuration Editor (see Section 13.4.3). The configuration is done in the same way as for

the Coordinator as either a fixed channel (in the range 11-26) or a set of channels to scan. Thus, the device listens for beacons in the relevant channel(s).

A beacon filter can be optionally introduced using the function **zps_bAppAddBeaconFilter()** to allow only beacons from networks of interest to be considered - beacons can be filtered on the basis of PAN ID, Extended PAN ID, LQI value, and device joining status/capacity (see Appendix B.4, Beacon filtering guidelines).

On completion of this search, the subsequent actions depend on the pre-set value of the 64-bit Extended PAN ID (EPID), which is set via the advanced device parameter *APS Use Extended PAN ID*in the ZPS Configuration Editor (see Section 13.4.4):

- If the pre-set EPID value is non-zero, this value identifies a specific network to join (assuming the Coordinator has been pre-set with the same EPID - see Section 6.1.1). Provided that a network with this EPID has been discovered in the search, the device attempts to join this network as described in Submits a join request to network (therefore bypassing the steps listed in Selects a network to join).
- If the pre-set EPID value is zero, the results of the search are reported in a zps_EVENT_NWK_DISCOVERY_COMPLETE stack event, which contains details of the networks discovered (see Section 6.2.1). The device must then select a network to join, as described in the following section.

Parent topic: Starting Routers and End Devices

Selects a network to join On the basis of the results in zps_EVENT_NWK_DISCOVERY_COMPLETE, the application must select a network which the device will attempt to join. The search results contain a recommended network, selected as the first ZigBee PRO network detected that allows nodes to join. The application is, however, free to choose another network, where this choice may be based on LQI value (detected signal strength).

Parent topic: Starting Routers and End Devices

Submits a join request to network Once the device identifies a network to join, a request to join the network must be submitted. If a non-zero pre-configured EPID has been set (see above), this join request is submitted automatically, otherwise the function **zps_eAplZdoJoinNetwork()** must be called to submit the request. The outcome of this request is reported in one of the following stack events on the requesting device:

- zps_EVENT_NWK_JOINED_AS_ROUTER (if joined as Router)
- zps_EVENT_NWK_JOINED_AS_ENDDEVICE (if joined as End Device)
- zps_EVENT_NWK_FAILED_TO_JOIN (if failed to join)

In the case of success, the above stack event contains the 16-bit network address that the network has allocated to the local device. In addition, the event zps_EVENT_NWK_NEW_NODE_HAS_JOINED is generated on the parent.

If the case of failure, the device can attempt another join by calling **zps_eAplZdoJoinNetwork()** with a different result reported in the zps_EVENT_NWK_DISCOVERY_COMPLETE event.

Parent topic: Starting Routers and End Devices

Records the network's EPID for application use The function **zps_eAplAibSetApsUseExtendedPanId()** may now be used to create a persistent record of the EPID of the network that the node has joined (it is necessary to first obtain the EPID value using the functions **zps_pvAplZdoGetNwkHandle()** and **zps_u64NwkGetEpid()**). If this EPID record is created, the node automatically continues in the network following a reset without explicitly rejoining.

Parent topic: Starting Routers and End Devices

Router accepts join requests from other devices (if enabled) A Router may now allow other devices (Routers and End Devices) to join it as its children. The number of (direct) children of the Router is limited by the maximum number of neighbors for the node, which is pre-set via the advanced network parameter *Active Neighbor Table Size*and *Child Table Size*in the ZPS Configuration Editor (see Section 13.4.4).

Note: The initial 'permit joining' status is pre-set via the Router parameter Permit Joining Time in the ZPS Configuration Editor. If this is initially disabled, the Router may not accept children until joining has been enabled using zps_eAplZdoPermitJoining(). However, the 'permit joining' status is ignored during a join in which the pre-set EPID on the joining device is non-zero and during any rejoin (see Section 6.6.2). The above function can be used at any time to allow joinings for a limited time-period or indefinitely, and can also be used to disable joinings.

Once a node has joined the network, each endpoint application on the node is next likely to search for compatible endpoints on remote nodes with which it can communicate, as described in Section 6.2.2.

Note: A network can be set up such that an End Device or Router joins a particular parent node. The required configuration and function calls to employ predetermined parents are described in Section 6.1.3.

Parent topic: Starting Routers and End Devices

Parent topic: Forming and joining a network

Pre-determined parents It is possible to force a parent (Router or the Coordinator) to accept certain nodes as its (direct) children. The function **zps_eAplZdoDirectJoinNetwork()** can be used on this parent to register a potential child node (with specified IEEE/MAC and network addresses) by adding this node to the Neighbor table - *never write to the Neighbor table directly*. The parent then regards this node as an orphaned child. This function should only be called when the parent node is fully up and running - that is, the node has been started as described in Section 6.1.1 or Section 6.1.2.

When one of the designated children is started, its application should call the function **zps_eAplZdoOrphanRejoinNetwork()** in order to attempt to join the network as if it were a previously orphaned node. This function will start the ZigBee PRO stack and attempt to join the network whose EPID has been pre-configured on the node (using the ZPS Configuration Editor). The function will only allow the node to join a parent that already has knowledge of the node (in the parent's Neighbor table).

Note:

- Note 1: When zps_eAplZdoOrphanRejoinNetwork() is used, the start-up procedure described in Section 6.1.2 is not applicable to the joining node and the function zps_eAplZdoStartStack() must not be explicitly called on the node.
- Note 2: When a node joins the network in this way, the 'permit joining' status on the parent is ignored.

If the node successfully joins the network (via the designated parent), the stack event zps_EVENT_NWK_NEW_NODE_HAS_JOINED is generated on the parent node and one of the following stack events is generated on the joined node:

- zps_EVENT_NWK_JOINED_AS_ROUTER (if joined as a Router)
- zps_EVENT_NWK_JOINED_AS_ENDDEVICE (if joined as an End Device)

These events contain the network address that the parent has allocated to the joined node.

If the join request is unsuccessful, the zps_EVENT_NWK_FAILED_TO_JOIN event is generated on the joining node.

Once the node has joined the pre-determined parent, the node is next likely to search for compatible endpoints on remote nodes with which it can communicate, as described in Section 6.2.2.

Parent topic: Forming and joining a network

Parent topic:*Application coding with ZigBee PRO APIs*

Discovering the network This section describes how to discover properties of the network, including general network properties, node addresses and features, and the services offered by nodes. The important task of finding nodes that can communicate with each other is described. Maintenance of the 'primary discovery cache' of a node is also described - this cache contains information about other nodes of the network (not all nodes will host a primary discovery cache - only the Coordinator and Routers are allowed to).

Obtaining network properties A 'network discovery' is implemented when the function **zps_eAplZdoStartStack()**is called to start the stack on an End Device or Router node (which needs to find a network to join). In addition, a network discovery can be explicitly started by calling the function **zps_eAplZdoDiscoverNetworks()**. For example, this function could be called if the initial network discovery did not find any suitable networks to join, in which case the function may be used to initiate a scan of previously unscanned channels (detailed in the stack event described below, resulting from the initial discovery).

Both of these function calls eventually result in the stack event zps_EVENT_NWK_DISCOVERY_COMPLETE on the End Device or Router, where this event reports the following properties of the discovered networks:

- Extended PAN ID
- ZigBee version
- ZigBee stack profile

This stack event also indicates the recommended network to join, which is taken to be the first ZigBee PRO network detected that is allowing nodes to join.

For information on joining a network, refer to Section 6.1.2.

Parent topic: Discovering the network

Finding compatible endpoints An endpoint on a newly joined node must find compatible endpoints on remote nodes with which to communicate. The decision of whether a remote endpoint is compatible is based on the endpoint properties stored in its Simple descriptor, notably the input/ output clusters supported.

The endpoint application can discover compatible nodes by sending out a Match_Desc_req request identifying the required clusters. This request is submitted by calling the function **zps_eAplZdpMatchDescRequest()**, which allows the request to be sent as a broadcast to all nodes or as a unicast to a particular node (the sending node may already have a record of the network nodes and their addresses, as each node automatically announces itself in a broadcast when it joins the network). The

request is sent in an APDU (Application Protocol Data Unit) which must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()**.

A receiving endpoint which satisfies the supplied criteria replies to the request with a Match_Desc_rsp response which, when received, must be collected on the requesting node using the function **ZQ_bZQueueReceive()**. The requesting application may bind to a compatible endpoint (see Section 5.4) and communicate with the endpoint using binding or addressing (see Section 5.5).

Parent topic: Discovering the network

Obtaining and maintaining node addresses The addresses of network nodes are needed in order to access node information (see Section 6.2.4), send data from one node to another (see Section 6.5) and bind nodes together (see Section 6.4). In most of these operations, an application can specify either 64-bit IEEE/MAC addresses or 16-bit network addresses, but the ZigBee PRO stack always works with network addresses. If the IEEE address (rather than the network address) of a remote node is specified by the application, the network address must still be available to the stack in an Address Map - see below.

The IEEE address of a node is assigned at the time of device manufacture and is fixed, while its network address is dynamically allocated by its parent when the device joins the network (this address may change if the network is re-started or the device later leaves and rejoins the network). Functions are provided to obtain the IEEE address of a node given its network address or to obtain the network address given the IEEE address. Use of these functions is described in Section 6.2.3.1 and Section 6.2.3.2.

Note: The IEEE/MAC and network addresses of a node can be broadcast to all other nodes in the network using the function zps_eAplZdpDeviceAnnceRequest(). For example, this function would typically be called when the node joins or rejoins the network. The information is sent in a Device_annce announcement, which must be collected by the recipient nodes using the function ZQ_bZQueueReceive().

An Address Map table can be maintained on a node, where each entry of this table contains the pair of addresses for a remote node - the 64-bit IEEE/MAC address and 16-bit network address. In fact, the IEEE/MAC address is not directly stored in the Address Map table but in a MAC Address table - the Address Map table contains the index of this address in the MAC Address table. The Address Map is automatically updated by the stack when a Device_annce announcement is received from a remote node (described in the Note above), but you can also add an address-pair to this table using the function **zps_eAplZdoAddAddrMapEntry()**- *never write to the Address Map table directly*. The Address Map must be properly maintained if the application employs IEEE/MAC addresses to identify remote nodes. In addition, when application- level security (see Section 6.8) is used in sending data from one node to another, the Address Map on the sending node must contain an entry for the target node.

Obtaining IEEE address You may wish to obtain the IEEE address of the node with a given network address - for example, in order to know which physical node corresponds to a particular dynamically allocated network address.

The IEEE address of the local node can be obtained simply by calling the function **zps_u64AplZdoGetIeeeAddr()**

The IEEE address of a remote node can be obtained in either of two ways, depending on whether an entry for the node exists in the local Address Map table:

- The function **zps_u64AplZdoLookupIeeeAddr()** can be used to search the local Address Map table for the IEEE address which corresponds to a given network address.
- The required IEEE address can be obtained directly from the remote node by using the function **zps_eAplZdpIeeeAddrRequest()** to submit a request for the IEEE address of the node with a particular network address. This request, of type IEEE_addr_req, is sent in an APDU (Application Protocol Data Unit) which must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()**. The request details are specified through the structure zps_tsAplZdpIeeeAddrReq, which includes an option to also request the IEEE addresses of all the target node's children (if any). The results are reported in an IEEE_addr_resp response.

Parent topic: Obtaining and maintaining node addresses

Obtaining network address You may wish to obtain the network address of the node with a given IEEE address - for example, in order to know the network address that has been dynamically allocated to a particular physical node.

The network address of the local node can be obtained simply by calling the function**zps_u16AplZdoGetNwkAddr()**.

The network address of a remote node can be obtained in either of two ways, depending on whether an entry for the node exists in the local Address Map table:

- **zps_u16AplZdoLookupAddr()** can be used to search the local Address Map table for the network address which corresponds to a given IEEE address.
- The required network address can be obtained directly from within the network by using the function **zps_eAplZdpNwkAddrRequest()** to submit a request for the network address of the node with a particular IEEE address. This request can be either unicast or broadcast, as follows:
  - Unicast to another node that will 'know' the required network address (this may be the parent of the node of interest or the Coordinator)
  - Broadcast to the network

This request, of type NWK_addr_req, is sent in an APDU (Application Protocol Data Unit) which must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()**. The request details are specified through the structure zps_tsAplZdpNwkAddrReq, which includes an option to also request the network addresses of all the target node's children (if any). The results are reported in a NWK_addr_resp response.

Parent topic: Obtaining and maintaining node addresses

Parent topic: Discovering the network

Obtaining node properties Functions are provided to obtain information about the properties of network nodes. Much of this information is held on a node in special structures, referred to as descriptors. Five types of descriptor are used:

- Node descriptor
- Node Power descriptor
- Simple descriptor
- User descriptor
- Complex descriptor

In addition to the above, information can be obtained about the active endpoints, primary discovery cache and services of a node.

The required functions are detailed below. Functions are provided to obtain descriptors from the local node and from a remote node. When obtaining information from a remote node, the function sends a request in an APDU (Application Protocol Data Unit) which must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()**. The results of the request are reported in a response which must be collected using the function **ZQ_bZQueueReceive()**.

Note:

1. When obtaining a descriptor of a remote node, the request can be submitted to the node itself or to another node which may hold the required descriptor in its primary discovery cache.
2. The structures that contain the descriptors (referenced below) are described in *Section 7.2* and Section 8.2.1.
3. Where 64-bit IEEE/MAC addresses are used to identify remote nodes, the corresponding 16-bit network addresses must be available in the local Address Map - see Section 5.2.3.

Node descriptor The Node descriptor contains basic information about the node, such as its ZigBee node type and the radio frequency bands supported. The following functions can be used to obtain a Node descriptor:

- **zps_eAplAfGetNodeDescriptor()** obtains the Node descriptor of the local node. The result is stored in a structure of type zps_tsAplAfNodeDescriptor.
- **zps_eAplZdpNodeDescRequest()** requests the Node descriptor of a remote node. The result is stored in a structure of type zps_tsAplZdpNodeDescriptor.

Parent topic: Obtaining node properties

Power descriptor The Node Power descriptor contains information about the node's supported power sources and present power source. The following functions can be used to obtain a Power descriptor:

- **zps_eAplAfGetNodePowerDescriptor()** obtains the Node Power descriptor of the local node. The result is stored in a structure of type zps_tsAplAfNodePowerDescriptor.
- **zps_eAplZdpPowerDescRequest()** requests the Node Power descriptor of a remote node. The result is stored in a structure of type zps_tsAplZdpNodePowerDescriptor.

Note that elements of the Node Power descriptor can be set on the local node using the ZPS Configuration Editor.

Parent topic: Obtaining node properties

Simple descriptor There is a Simple descriptor for each endpoint on a node. The information in this descriptor includes the ZigBee device type supported by the endpoint as well as details of its input and output clusters. The following functions can be used to obtain a Simple descriptor:

- **zps_eAplAfGetSimpleDescriptor()** obtains the Simple descriptor of a particular endpoint on the local node. The result is stored in a structure of type zps_tsAplAfSimpleDescriptor.
- **zps_eAplZdpSimpleDescRequest()** requests the Simple descriptor of a particular endpoint on a remote node. The result is stored in a structure of type zps_tsAplZdpSimpleDescReq.

The returned Simple descriptor includes a list of input clusters and a list of output clusters of the endpoint.

When requesting a Simple descriptor from a remote node, if the cluster lists are long, the Simple descriptor may not fit into the APDU of the response. In this case, the returned Simple descriptor will contain incomplete cluster lists, but the remainder of the lists can be recovered using **zps_eAplZdpExtendedSimpleDescRequest()**.

It is also possible to search for nodes on the basis of certain criteria in the Simple descriptors of their endpoints - for example, search for endpoints which have a particular list of input clusters and/or output clusters. Such a search can be performed using the function **zps_eAplZdpMatchDescRequest()**. Use of this function is described in Section 6.2.2.

Parent topic: Obtaining node properties

User Descriptor The User descriptor is a user-defined character string, normally used to describe the node (for example, "Thermostat"). The maximum length of the character string is 16, by default. A node need not have a User descriptor - if it has one, this must be indicated in the Node descriptor. The following functions can be used to access a User descriptor:

- **zps_eAplZdpUserDescSetRequest()**sets the User descriptor of a remote node.
- **zps_eAplZdpUserDescRequest()**requests the User descriptor of a remote node. The result is stored in a structure of type zps_tsAplZdpUserDescReq.

The above functions can only be used to access the User descriptor of a non-NXP device (which supports this descriptor), since the storage of a User descriptor on an NXP device is not supported.

Parent topic: Obtaining node properties

Complex descriptor The Complex descriptor is an optional descriptor which contains device information such as manufacturer, model and serial number. The function **zps_eAplZdpComplexDescRequest()** allows the Complex descriptor of a remote node to be requested. However, the NXP ZigBee PRO stack does not support the functionality to produce a valid response and this function is provided only for compatibility with non-NXP products that do support the relevant functionality.

Parent topic: Obtaining node properties

Active endpoints An endpoint on the local node can be configured as enabled or disabled using the function **zps_eAplAfSetEndpointState()**. An enabled endpoint is described as 'active'. The current state of a local endpoint can be obtained using the function **zps_eAplAfGetEndpointState()**.

It is also possible to configure whether a local endpoint will be included in the results of network discovery operations, for example, when **zps_eAplZdpMatchDescRequest()**is called. The 'discoverable' state of a local endpoint can be set using the function **zps_eAplAfSetEndpointDiscovery()**, while this state can be obtained using the function **zps_eAplAfGetEndpointDiscovery()**.

A list of the active endpoints on a remote can be obtain using the function **zps_eAplZdpActiveEpRequest()**. This functions submits an Active_EP_req request to the target node, which replies with an Active_EP_rsp response. If the active endpoint list is too long to fit into the APDU of the response, the returned list will be incomplete. However, the remainder of the list can be recovered using the function **zps_eAplZdpExtendedActiveEpRequest()**. Note that an endpoint is included in the list only if it is active and discoverable.

Parent topic: Obtaining node properties

Primary discovery cache A ZigBee routing node (Router or the Coordinator) may be able to host a 'primary discovery cache'. This is a database, held in memory, containing 'discovery information' about a set of network nodes, normally children and possibly other descendant nodes. The information held about a node includes the node's addresses, descriptors (Node, Node Power, Simple) and its list of active endpoints. Remote nodes can then interrogate the primary discovery cache to obtain information about other nodes in the network.

Note: NXP nodes do not have the capability to hold a primary discovery cache, but functions are provided to interface with a primary discovery cache held on a node from another manufacturer.

The function **zps_eAplZdpDiscoveryCacheRequest()** allows nodes which hold a primary discovery cache to be detected. This function submits a Discovery_Cache_req request to the network. Nodes with a primary discovery cache reply with a Discovery_Cache_rsp response.

In addition, the function **zps_eAplZdpFindNodeCacheRequest()** can be used to search for nodes with a primary discovery cache that holds information about a particular node. This function submits a Find_node_cache_req request to the network. Nodes with the required node information in their caches reply with a Find_node_cache_rsp response.

Functions for storing node information in a primary discovery cache are described in Section 6.2.5.

Parent topic: Obtaining node properties

Servers A node can host one or more of the following 'servers' in a ZigBee PRO network:

- Primary Trust Centre
- Backup Trust Centre
- Primary Binding Table Cache
- Backup Binding Table Cache
- Primary Discovery Cache
- Backup Discovery Cache
- Network Manager

The function **zps_eAplZdpSystemServerDiscoveryRequest()** can be used to discover the servers hosted by other nodes in the network. The function broadcasts a System_Server_Discovery_req request to all nodes. A remote node replies with a System_Server_Discovery_rsp response containing a bitmap indicating the servers hosted by the node.

Parent topic: Obtaining node properties

Parent topic: Discovering the network

Maintaining a primary discovery cache Some routing nodes of a ZigBee PRO network may be capable of hosting a primary discovery cache, which contains 'discovery information' relating to other nodes in the network - see Primary Discovery Cache.

Note: NXP nodes do not have the capability to hold a primary discovery cache, but functions are provided to interface with a primary discovery cache held on a node from another manufacturer.

Functions are provided for storing the local node's 'discovery information' in another node's primary discovery cache (normally in the parent or another ascendant node). First of all, **zps_eAplZdpDiscoveryStoreRequest()** must be called to allocate memory space for this information in the remote node's cache. This function sends a Discovery_store_req request to the remote node, which replies with a Discovery_store_rsp response. The local node's information can then be stored in the remote node's primary discovery cache using the following functions (which all operate on a request/response basis):

- **Node descriptor:** Stored using **zps_eAplZdpNodeDescStoreRequest()**
- Power descriptor: Stored using **zps_eAplZdpPowerDescStoreRequest()**
- **Simple descriptor:** Stored using **zps_eAplZdpSimpleDescStoreRequest()**
- **Active endpoints list:** Stored using **zps_eAplZdpActiveEpStoreRequest()**

A node's information can be removed from a primary discovery cache using the function **zps_eAplZdpRemoveNodeCacheRequest()**. This function can be called on the local node to remove a third node's information from the primary discovery cache of a remote node.

Parent topic: Discovering the network

Discovering Routes The route from one network node to another can be pre-established by implementing a route discovery. As a result, each routing node along the route will contain a Routing table entry for the destination node, where this entry consists of the destination address and the 'next hop' address. Routing and route discovery are fully introduced in Section 3.5.

Two functions are provided in the ZigBee PRO API to initiate route discoveries:

- **zps_eAplZdoRouteRequest()** can be used to establish a route from the local node to a specific destination node. This kind of end-to-end route discovery is outlined in Section 3.5.2.

- **zps_eAplZdoManyToOneRouteRequest()** can be used on a 'concentrator' node to implement a 'many-to-one' route discovery back to itself. The result is that Routing tables in routing nodes within a certain radius of the concentrator will acquire entries with the concentrator as the destination. Many-to-one routing is outlined in Section 3.5.3.

Parent topic: Discovering the network

Parent topic:*Application coding with ZigBee PRO APIs*

Managing group addresses A 'group address' is a concept that simplifies data transfers (see Section 6.5) to multiple nodes/endpoints. It is a collective 16-bit address which refers to a group of destination endpoints (that may be located on different nodes). So, for example, when a group address is specified as the destination address for a data transfer, the data will be delivered to all the nodes/endpoints in the associated group. It is the responsibility of the wireless network application to allocate and manage group addresses on a network-wide basis.

A node which is to receive group-addressed communications must have a Group Address table. This table contains information about all the groups to which endpoints on the node belong - that is, each group address and the associated local endpoint numbers. The table is consulted on receiving a data packet with a group address - if the group address exists in the table, the packet is passed to the corresponding endpoint(s).

A Group Address table is created on a node using the ZPS Configuration Editor. The table can then be maintained by the application as follows:

- An endpoint can be added to a group by calling the function **zps_eAplZdoGroupEndpointAdd()** on the local node (which contains the endpoint).
- An endpoint can be removed from a group by calling the function **zps_eAplZdoGroupEndpointRemove()** on the local node (which contains the endpoint). Alternatively, **zps_eAplZdoGroupAllEndpointRemove()** can be used to remove a specified local endpoint from all groups to which it belongs.

The group addresses used in a network are defined by the application developer.

Parent topic:*Application coding with ZigBee PRO APIs*

Binding For the purpose of data communication between applications running on different nodes, it may be useful to 'bind' the relevant source and destination endpoints. When data is subsequently sent from the source endpoint, it is automatically routed to the bound destination endpoint(s) without the need to specify a destination address. For example, a binding could be created between the temperature sensor endpoint on a thermostat node and the switch endpoint on a heating controller node. Details of a binding are held in a Binding table on the source node. Binding is introduced more fully in Section 3.6.2, where bindings are one-to-one, one-to-many or many-to-one.

This section describes setting up a Bind Request Server and how to bind together two nodes, as well as how to unbind them. Access to the Binding tables is also described.

Note: Where 64-bit IEEE/MAC addresses are used to identify remote nodes, the corresponding 16-bit network addresses must be available in the local Address Map - see Section 5.2.3.

Setting up bind request server A Bind Request Server must be set up on each device that will be the source node of a bound data transfer. This server manages a bound data transfer so that application processing is not blocked by concurrent requests for transmissions to the multiple destinations of the transfer. It does this by limiting the number of destinations and inserting a time delay between consecutive transmissions of a bound transfer.

Note: The bound server can only handle one bound request at a time. The application must wait for the confirmation from the first bound request before attempting to send a second bound request.

The server is configured in the ZPS Configuration Editor (introduced in *Chapter 13*). Two parameter values must be set:

Simultaneous requests This refers to the maximum number of destinations for a bound data transfer. The value set must be less than or equal to the value of the ZigBee network parameter *Maximum Number of Simultaneous Data Requests*or *Maximum Number of Simultaneous Data Requests with Acks*, described in Section 11.7.

Parent topic: Setting up bind request server

Time interval This refers to the time interval between consecutive transmissions to the different destinations of a bound data transfer and is measured in milliseconds.

In the ZPS Configuration Editor, these parameters are accessed by clicking on **Bind Request Server** under **ZDO Configuration** for the device (the parameters appear in the **Properties** tab of the bottom pane).

Note: The bound server can only handle one bound request at a time. The application must wait for the confirmation from the first bound request before attempting to send a second bound request.

Parent topic: Setting up bind request server

Parent topic: Binding

Binding endpoints An endpoint on the local node can be bound to one or more endpoints on remote nodes using the following functions:

- zps_eAplZdoBind() creates a one-to-one binding to a single remote endpoint.
- zps_eAplZdoBindGroup() creates a one-to-many binding for which the destination endpoints are specified via a group address (refer to Section 5.3).

The function zps_eAplZdpEndDeviceBindRequest() is also provided, which allows an endpoint on one End Device to be bound to an endpoint on another End Device via the Coordinator. This function must be called on both End Devices, where the function call would typically be triggered by a user action such as pressing a button on the node. The function submits an End_Device_Bind_req request to the Coordinator, which replies with an End_Device_Bind_rsp response. The stack will then automatically update the Binding tables on the End Devices (as the result of bind requests from the Coordinator), and these updates will be indicated by a zps_EVENT_ZDO_BIND event on each of the End Devices.

Parent topic: Binding

Unbinding endpoints Bindings can be removed using the following functions:

- Two endpoints previously bound using **zps_eAplZdoBind()** can be unbound using the function **zps_eAplZdoUnbind()**.
- Endpoints previously bound using **zps_eAplZdoBindGroup()** can be unbound using the function **zps_eAplZdoUnbindGroup()**.

Parent topic: Binding

Accessing binding tables Information about established bindings is held in Binding tables on the relevant nodes. Normally, a Binding table is held on a node which contains at least one source endpoint for a binding - thus, the table includes entries for all bindings which involve source endpoints on the local node. Alternatively, the Binding table entries for a particular source node can be held in a primary Binding table cache on the node's parent or another ascendant node.

However, if a primary Binding table cache exists on an ascendant node, a source node can opt out of membership of this table by calling the function **zps_eAplZdpBindRegisterRequest()** to indicate that the source node will store its own Binding table entries locally.

Functions are provided which allow Binding tables to be remotely accessed and modified. These functions are particularly useful in implementing a commissioning tool application.

A binding can be remotely created or removed by requesting a modification to the relevant Binding table on a remote node. The remote Binding table may be a primary Binding table cache or the source node's local Binding table, which is relevant for the particular binding.

- The function **zps_eAplZdpBindUnbindRequest()** can be used to request that a new binding is added to a remote Binding table. The addition of this binding is signaled by a zps_EVENT_ZDO_BIND event on the remote node.
- The function **zps_eAplZdpBindUnbindRequest()** can also be used to request that an existing binding is removed from a remote Binding table. The removal of this binding is signaled by a zps_EVENT_ZDO_UNBIND event on the remote node. A Binding table entry can also be removed locally using the function **zps_eAplAibRemoveBindTableEntryForMacAddress()**, which requests that the entry containing a particular IEEE/MAC address is deleted.

In addition, binding entries in a remote primary Binding table cache can be modified using the function **zps_eAplZdpReplaceDeviceRequest()**, to replace an IEEE/MAC address and/or endpoint number. This operation works on a 'search and replace' basis in the Binding table, and the address/endpoint number to be replaced could occur in the source or destination of one or more table entries.

The function **zps_eAplZdpMgmtBindRequest()** is also provided, which can be used to request the Binding table of a remote node.

Parent topic: Binding

Parent topic:*Application coding with ZigBee PRO APIs*

Transferring data This section describes how to send data to a remote node and receive the data at the destination. The data polling method is also described, which is used by an End Device to obtain data that arrives at its parent while the End Device is asleep.

Sending data Data is sent across the wireless network in an Application Protocol Data Unit (APDU). Before calling the function to send the data, an APDU instance must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then populated with data using the PDUM function **PDUM_u16APduInstanceWriteNBO()**.

There are five ways to send data to one or more remote nodes:

- **Unicast:** Sending data to a single destination endpoint
- **Broadcast:** Sending data to (potentially) all endpoints
- **Group Multicast:** Sending data to a group of endpoints
- **Bound Transfer:** Sending data to bound endpoints
- **Inter-PAN Transfer:** Sending data to another ZigBee PRO network

These methods are described in the sub-sections below. However, in all cases except the inter-PAN transfer, a general function **zps_eAplAfApsdeDataReq()** can be used which imposes no restrictions on the destination address, destination cluster and destination endpoint number - these destination parameters do not need to be known to the stack or defined in the ZPS configuration.

Note:

1. In all cases, once the data packet has been successfully sent, a 'DATA_CONFIRM' stack event is generated. When sending data to one or more individual nodes (not broadcasting), this event is generated after a MAC-level acknowledgment has been received from the 'next hop' node.
2. Where 64-bit IEEE/MAC addresses are used to identify remote nodes, the corresponding 16-bit network addresses must be available in the local Address Map - see Section 5.2.3.

Unicast A unicast is a data transmission to a single destination - in this case, a single endpoint. The destination node for a unicast can be specified using the network address or the IEEE/MAC address of the node:

- **zps_eAplAfUnicastDataReq()** is used to send a data packet to an endpoint on a node with a given network address.
- **zps_eAplAfUnicastIeeeDataReq()** is used to send a data packet to an endpoint on a node with a given IEEE/MAC address.

Neither of these functions provide any indication that the data packet has been successfully delivered to its destination. It is possible that a unicast packet will not reach its destination because the packet is lost - for example, it becomes caught in a circular route. However, equivalent functions are available which request the destination node to provide an acknowledgment of data received - these 'with acknowledgment' functions are **zps_eAplAfUnicastAckDataReq()** and **zps_eAplAfUnicastIeeeAckDataReq()**, requiring network and IEEE/MAC addresses respectively. These functions request end-to-end acknowledgments which, when received, generate zps_EVENT_APS_DATA_ACK events (note that the 'next hop' zps_EVENT_APS_DATA_CONFIRM events will also be generated). A timeout of approximately 1600 ms is applied to the acknowledgments. If an acknowledgment has not been received within the timeout period, the data is re-sent, and up to 3 more re-tries can subsequently be performed before the data transfer is abandoned completely (which occurs approximately 3 seconds after the initial send).

Note: If a message is unicast to a destination for which a route has not already been established, the message will not be sent and a route discovery will be performed instead. If this is the case, the unicast function will return zps_NWK_ENUM_ROUTE_ERROR. The application must then wait for the stack event zps_EVENT_NWK_ROUTE_DISCOVERY_CONFIRM(success or failure) before attempting to re-send the message by calling the same unicast function again.

Unicasts from sleepy nodes To allow a unicast acknowledgment to be received as described above, the source node must remain awake for a time equal to the timeout period. On a battery-powered node which sleeps, the use of acknowledgments and retries may not be desirable from a power-saving point of view. In this case, acknowledgments should not be used, but it is good practice for the application to monitor the route to a remote node by periodically attempting to read an attribute on the node and wait for a response. If the response is not observed within a pre-defined time then the application should take one of the actions listed below, depending on whether the source node is an End Device or Router.

- If an End Device, the application should notify the parent node about the routing problem by sending it a unicast network status command using the function **zps_vNwkSendNwkStatusCommand()**, with the status as "No Route Available (0x00)"
- If a Router, the application should initiate an explicit route discovery to the destination node by calling the function **zps_eAplZdoRouteRequest()**

Parent topic: Unicast

Fragmenting large unicast packets The unicast 'with acknowledgment' functions, **zps_eAplAfUnicastAckDataReq()** and **zps_eAplAfUnicastIeeeAckDataReq()**, also allow a

large data packet to be sent that may be fragmented into multiple messages during transmission. Application design issues concerned with fragmented data transfers are outlined in Fragmented data transfers.

Parent topic: Unicast

Parent topic: Sending data

Broadcast A broadcast is a data transmission to all network nodes, although it is possible to select a subset of nodes. The following destinations are possible:

- All nodes
- All nodes for which 'receiver on when idle' - these include the Coordinator, Routers and non-sleeping End Devices
- All Routers and the Coordinator

The function **zps_eAplAfBroadcastDataReq()** is used to broadcast a data packet. It is possible to specify a particular destination endpoint on the nodes (the same endpoint number for all recipient nodes) or all endpoints. Following this function call, the packet may be broadcast up to four times (in addition, the packet may be subsequently re-broadcast up to four times by each intermediate routing node).

Parent topic: Sending data

Group multicast A group multicast is a data transmission which is intended for a selection of network nodes or, more specifically, a selection of endpoints on these nodes. The set of destination endpoints must be pre-assembled into a group with an associated 'group address', as described in Section 5.3.

The function **zps_eAplAfGroupDataReq()** is used to send a data packet to the group of endpoints with a given group address. In practice, the data packet is broadcast to all nodes in the network and it is the responsibility of each recipient node to determine whether it has endpoints in the target group (and therefore whether the packet is of interest).

Parent topic: Sending data

Bound transfer A data packet can be sent from an endpoint to all the remote endpoints with which the source endpoint has been previously bound (see Section 5.4). The function **zps_eAplAfBoundDataReq()** is used to implement this type of data transfer. This method provides an alternative to a group multicast (see Section 5.5.1.3) for sending data to selected endpoints.

An equivalent to the above function is provided which also requests an 'end-to-end' acknowledgment from the destination - **zps_eAplAfBoundAckDataReq()**. If an acknowledgment has not been received within approximately 1600 ms of the initial request, the data is re-sent, with up to 3 more subsequent re-tries before the data transfer is abandoned completely.

zps_eAplAfBoundAckDataReq() also allows a large data packet to be sent that may need to be fragmented into multiple messages during transmission. Application design issues concerned with fragmented data transfers are outlined in Fragmented data transfers.

Following a call to one of the above bound transfer functions, a deferred zps_EVENT_BIND_REQUEST_SERVER event is generated on the sending node. This event summarizes the status of the transmission (see Section 7.2.2.21), including the number of bound endpoints for which the transmission failed. The event is generated only after receiving MAC-level acknowledgments from the 'next hop' nodes or, if requested, after receiving end-to-end acknowledgments from the destination nodes.

Note: In the case of a bound transfer, the 'next hop' zps_EVENT_APS_DATA_CONFIRM events and 'end-to-end' zps_EVENT_APS_DATA_ACK events are consumed and do not reach the application.

Parent topic: Sending data

Inter-PAN transfer A data packet can be sent to nodes in other IEEE 802.15.4 networks - this is referred to as an inter-PAN transfer or transmission. Typically, this mechanism could be used to send information to optional low-cost devices that are not part of the local network. Note that no security (encyption/decryption) can be applied to inter-PAN transfers and only one application on a device can perform inter-PAN transmissions. The inter-PAN messages are not forwarded and so will only be received by nodes within direct radio range of the transmitter.

The inter-PAN feature is enabled via the ZPS Configuration Editor. The *Inter PAN*value is set to true in the APS Layer Configuration section of the Advanced Properties for the device.

The function **zps_eAplAfInterPanDataReq()** is used to request an inter-PAN transmission. This function requires the destination(s) for the transfer to specified:

- Single destination node in a specific network (PAN ID and node address must be specified)
- Multiple destination nodes in a specific network

(PAN ID and a group address for the nodes must be specified)

- All nodes in a specific network

(PAN ID and broadcast address of 0xFFFF must be specified)

- All nodes in all reachable networks

(broadcast PAN ID and broadcast address, both of 0xFFFF, must be specified)

After successfully sending the data packet, the stack will generate the event zps_EVENT_APS_INTERPAN_DATA_CONFIRM (for a single destination, this event is generated once the 'next hop' acknowledgment has been received).

A destination endpoint is not specified for this type of data transfer but a cluster must be specified for the destination. On receiving the data packet, the recipient node will automatically pass the packet to the endpoint which supports the given cluster (see Section 5.5.2).

Note:

1. In the case of a data packet received from another network by means of an inter-PAN transfer, the **zps_EVENT_APS_INTERPAN_DATA_INDICATION** stack event will be generated. The data packet will be passed to the endpoint which supports the specified cluster. The application must always handle these inter-PAN packets and release the APDU instances (see below). The event will only be generated if the inter-PAN feature has been enabled via the ZPS Configuration Editor. If an application transmits inter-PAN messages but does not need to receive them, the application must enable inter-PAN in the ZPS Configuration Editor and handle any **zps_EVENT_APS_INTERPAN_DATA_INDICATION** events by releasing the APDU instances.
2. In the case of the arrival of a response packet which is destined for the ZDO, a **zps_EVENT_AF_DATA_INDICATION** stack event will be generated with a destination endpoint of 0. It will be necessary for the application to call the function **zps_bAplZdpUnpackResponse()** to extract the response data from the event.

Parent topic: Sending data

Parent topic: Transferring data

Receiving data When a data packet (sent using one of the methods described in Section 6.5.1) is received by the destination node, it is put into a message queue. A zps_EVENT_AF_DATA_INDICATION stack event is generated on the destination node to indicate that a data packet has arrived (the destination endpoint is indicated in this event). The packet must then be collected from the message queue using the function **ZQ_bZQueueReceive()**.

Note:

- **Note 1**: In case a data packet is received from another network by means of an inter-PAN transfer, the zps_EVENT_APS_INTERPAN_DATA_INDICATION stack event is generated. The data packet is passed to the endpoint that supports the specified cluster. The application must always handle these inter-PAN packets and release the APDU instances (see below). The event will only be generated if the inter-PAN feature has been enabled via the ZPS Configuration Editor. If an application transmits inter-PAN messages but does not need to receive them, the application must enable inter-PAN in the ZPS Configuration Editor and handle any zps_EVENT_APS_INTERPAN_DATA_INDICATION events by releasing the APDU instances.
- **Note 2**: In the case of the arrival of a response packet which is destined for the ZDO, a zps_EVENT_AF_DATA_INDICATION stack event is generated with a destination endpoint of 0. It is necessary for the application to call the function zps_bAplZdpUnpackResponse() to extract the response data from the event.

An End Device that is asleep would be unable to receive a data packet directly, so the data is buffered by its parent for collection later. The End Device must explicitly request this data, once awake. This method of receiving data is called data polling and is described in Section 6.5.3.

Once a data packet has been collected from a message queue, the data can be extracted from the APDU instance using the PDUM function **PDUM_u16APduInstanceReadNBO()**. The APDU instance must then be released using the PDUM function **PDUM_eAPduFreeAPduInstance()**.

Parent topic: Transferring data

Polling for Data In the case of an End Device which is capable of sleeping, messages are not delivered directly to the device, since it may be asleep when the messages arrive. Instead, the messages are temporarily buffered by the End Device's parent. Once awake, the End Device can then ask or 'poll' its parent for data.

Note: End Devices that are not enabled for sleep can receive messages directly and therefore do not need to poll. An End Device is pre-configured as either sleeping or non-sleeping via the End Device parameter Sleeping in the ZPS Configuration Editor (see Section 12.4.2).

Data polling is performed using the function **zps_eAplZdoPoll()** in the End Device application. This function requests the buffered data and should normally be called immediately after waking from sleep. If the poll request is successfully sent to the parent, a zps_EVENT_NWK_POLL_CONFIRM stack event will occur on the End Device. The subsequent arrival of data from the parent is indicated by the stack event zps_EVENT_AF_DATA_INDICATION. Any messages forwarded from the parent should then be collected from the relevant message queue using the function **ZQ_bZQueueReceive()**, just as for normal data reception described in Section 6.5.2.

Application design issues concerned with transferring data to a sleeping End Device are outlined in Appendix B.2.

Parent topic: Transferring data

Security in data transfers The 'send data' functions for unicast, broadcast, group transfer and bound transfer contain a parameter to select the required security setting for the protection of the sent message. In the NXP ZigBee PRO software, there are currently three security options, as follows:

- No security
- Network-level security
- Application-level security

Application-level security is only available for unicast and bound transfers, while network-level security is available for all transfer types except inter-PAN transfers.

Network-level and application-level security are detailed in Section 5.8.

Note:

1. No security is available for inter-PAN transfers (to other networks).
2. When application-level security is used in sending data, the IEEE/MAC address and network address of the target node must be available through the local Address Map table - see Section 5.2.3.

Parent topic: Transferring data

Parent topic:*Application coding with ZigBee PRO APIs*

Leaving and rejoining the network This section describes how a node may leave the network and later rejoin either the same network or a different network.

Leaving the network A node may leave the network intentionally or unintentionally:

- The node may be intentionally (and temporarily) removed from the network for maintenance work, such as the replacement of batteries.
- The node may unintentionally leave the network due to unforeseen circumstances, such as a broken radio link with its parent (an obstacle may have been introduced into the path of the signal).

A node can be intentionally removed from the network using the function **zps_eAplZdoLeaveNetwork()**, which issues a leave request. The target node can be the requesting node itself or a child of the requesting node. The application may be designed to call this function when a button is pressed on the requesting node.

When calling **zps_eAplZdoLeaveNetwork()**:

- You can specify whether the children of the leaving node should also be requested to leave the network. If this is the case, the leaving node will first automatically call **zps_eAplZdoLeaveNetwork()** for each of its children.
- You can specify whether the leaving node should immediately attempt to rejoin the same network after leaving (see Section 5.6.2).

The stack event zps_EVENT_NWK_LEAVE_INDICATION is generated on the node which has been requested to leave (this event is also generated when a neighboring node has left the network). Once a node has been successfully removed from the network as the result of a call to **zps_eAplZdoLeaveNetwork()**, the stack event zps_EVENT_NWK_LEAVE_CONFIRM is generated on the requesting node.

The function **zps_eAplZdpMgmtLeaveRequest()** is also provided which can be used to request a remote node to leave the network.

By default, a Router will always act on leave request messages. However, it may be desirable for a Router to ignore leave request messages in order to prevent a rogue node from disrupting the network. If the function **zps_vNwkNibSetLeaveAllowed()** is called with the *bLeave*parameter as FALSE, the Router will ignore network leave requests. End Devices always act on leave requests from their parent and ignore leave requests from other nodes.

Alternatively, a callback function can be defined that is invoked when a leave request is received, where this function determines whether the leave request is to be obeyed

- this decision may depend on where the leave request came from. The callback function is registered using **zps_eAplZdoRegisterZdoLeaveActionCallback()**- refer to the description of this function for details of the callback function.

Parent topic: Leaving and rejoining the network

Rejoining the network A node may leave its network - for example, by:

- losing radio contact with its parent - the stack on the 'orphaned' node will detect this loss and automatically attempt to rejoin the network
- calling **zps_eAplZdoLeaveNetwork()** - the node will automatically attempt to rejoin the network if an immediate rejoin has been requested in the function call (although the node can be configured to always rejoin the network following a leave, using the function **zps_vNwkNibSetLeaveRejoin()**)

If the node successfully rejoins the network, the stack event zps_EVENT_NWK_NEW_NODE_HAS_JOINED is generated on the parent node and one of the following stack events is generated on the joined node:

- zps_EVENT_NWK_JOINED_AS_ROUTER (if joined as a Router)
- zps_EVENT_NWK_JOINED_AS_ENDDEVICE (if joined as an End Device)

These events contain the network address that the parent has allocated to the joined node (this may be different from the network address that the node previously had).

If the join request is unsuccessful, the zps_EVENT_NWK_FAILED_TO_JOIN event is generated on the requesting node.

If an automatic rejoin has failed or has not been requested, the function **zps_eAplZdoRejoinNetwork()** can be used to request a rejoin (this function must be called on the node that needs to rejoin). The application may be designed to call this function when a button is pressed on the node. The result of this function call will be indicated by means of the above events.

The function **zps_eAplZdpMgmtDirectJoinRequest()** is also provided which submits a request to a remote parent to allow a particular node to join it. In addition, the function **zps_eAplZdpMgmtPermitJoiningRequest()** is provided which allows joining to be enabled/disabled on a remote node.

Note:

1. When a device rejoins a network, the 'permit joining' status on the potential parent is ignored.
2. When a device joins the network, its application may call zps_eAplZdpDeviceAnnceRequest() to announce the device's membership and network address to the rest of the network. The information is sent in a Device_annce announcement, which must be collected by the recipient nodes using the function ZQ_bZQueueReceive().

Note: (Caution): If a node rejoins the same secured network but its stack context data was cleared before the rejoin (by calling NvErase()), data sent by the node will be rejected by the destination node since the frame counter has been reset on the source node. Therefore, you are not recommended to clear the stack context data before a rejoin. For more information and advice, refer to Appendix B.3.

Parent topic: Leaving and rejoining the network

Parent topic:*Application coding with ZigBee PRO APIs*

Return codes and extended error handling When a ZigBee PRO API function is called, a code is normally returned on completion of the function to indicate the outcome. This code is taken from one of the following:

- zps_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

An extended error handling mechanism can be optionally implemented which allows more detail to be obtained about certain errors that can occur during function execution. The particular errors are:

- 0xA3: zps_APL_APS_E_ILLEGAL_REQUEST
- 0xA6: zps_APL_APS_E_INVALID_PARAMETER
- 0xC2: zps_NWK_ENUM_INVALID_REQUEST

The extended error codes are listed and described in Section 11.2.5.

In order to implement the extended error handling mechanism, you must register a callback function using the function **zps_vExtendedStatusSetCallback()**. This registration function must be called before invoking the first API function for which extended error handling is required. The registered callback function will then be invoked during execution of the API function if one of the above errors occurs. The callback function will return an extended error code (from those listed in Section11.2.5) but the API function will return only the basic error code.

Parent topic:*Application coding with ZigBee PRO APIs*

Implementing ZigBee security The NXP ZigBee PRO APIs allow ZigBee security to be implemented, which applies key-based encryption to communications between network nodes. The message frame content generated at the NWK layer and higher is encrypted using 128-bit AES-based encryption (see Section 2.10). The NWK payload of the frame is encrypted, and the NWK header and payload are integrity-protected with a 32-bit Message Integrity Code (MIC).

This section describes security in a network with centralized security that is managed by a single Trust Centre, which is usually the Coordinator node. A distributed security scheme can alternatively be used and this is described in Section 6.10.2.

The sub-sections below deal with the following topics:

- Security levels - see Section 6.8.1
- Security keys - see Section 6.8.2
- Security set-up - see Section 6.8.3
- Security key modification - see Section 6.8.4

Security levels Two types or levels of security can be applied in a ZigBee network:

- **Network-level security**: This uses a 'network key' which is common throughout the network and is used to encrypt/decrypt all communications between all nodes. The network key is randomly generated by the Trust Centre before any nodes join the network. Setting up network-level security is described in Section 5.8.3.1.
- **Application-level security**: This uses an application 'link key' which is used (in addition to the network key) to encrypt/decrypt communications between a pair of nodes. This link key may be unique for a pair of nodes. Setting up application-level security is described in Section 5.8.3.2.

The encryption keys for these security levels are described in Section 5.8.2.

Parent topic: Implementing ZigBee security

Security key types The different ZigBee security keys are summarized in the Table, Table 1 and described in more detail below.

When a node joins the network, the Trust Centre must pass the network key to the joining node, so that the node can participate in network-level encrypted communications with existing network nodes. The network key must itself be protected by encryption when it is passed to the

joining node. For this encryption, a pre-configured link key is used, which is known by both the Trust Centre and the joining node. This can be a global link key or a unique link key:

- **Pre-configured global link key**: This link key is the same for all nodes in the network. It may be ZigBee-defined key or manufacturer-defined:
  - The ZigBee-defined key (known as the ZigBee "09" key) allows nodes from different manufacturers to join the network.
  - A manufacturer-defined key allows only nodes from the specific manufacturer to join the network.
- **Pre-configured unique link key**: This link key is an exclusive key for the Trust Centre and joining node. In this case, every node has a different link key.

The pre-configured link key must be pre-programmed into the relevant nodes either in the factory or during commissioning.

The network-level security set-up process is described in Section 6.8.3.1. The active network key can subsequently be changed at any time, as described in Section 6.8.4.

Once network-level security is set up, application-level security can be set up for more secure communications - this level of security is applied on top of network-level security. If application-level security is required between two nodes then a link key must be established for the nodes. This key can be any of the following:

- **Pre-configured global link key** (as detailed above): This is for communications between the Trust Centre and all other nodes
- **Pre-configured unique link key** (as detailed above): This is for communications between the Trust Centre and one other node
- **Trust Centre Link Key (TCLK)**: This is for communications between the Trust Centre and one other node. It is randomly generated by the Trust Centre and passed to the relevant node, for which it is encrypted with the network key and, if it exists, the pre-configured unique link key for the node. The TCLK is then used to encrypt all subsequent communications with the Trust Centre, replacing any pre-configured link key. The application should hold on to the pre- configured key, in case it needs to be reinstated in the future (for example, for a re-join).
- **Application link key:** This is for communications between a pair of nodes that does not include the Trust Centre. It is requested from the Trust Centre by one of the two nodes. The Trust Centre randomly generates the key and associates it with the IEEE/MAC addresses of the two nodes. The Trust Centre passes the key to each node, for which it is encrypted with the network key and, if it exists, the pre-configured unique link key for the node.

The application-level security set-up process is described in Section 6.8.3.2.

ZigBee security key summary

Security Key	Description
Netwo level Security	
Network key	- Essential key used to encrypt communications between all nodes of the network- Randomly generated by the Trust Centre- Distributed to joining nodes, encrypted with a pre-configured link key (see below)
Applic level Security	
Global link key (pre-configu	- Used between the Trust Centre and all other nodes- Pre-configured in all nodes (unless a unique link key is pre-configured - see below)- Also used in joining to encrypt network key transported from Trust Centre to joining node- If ZigBee-defined, allows nodes from all manufacturers to join the network- If manufacturer-defined, allows only nodes from one manufacturer to join the network- Touchlink Pre-configured Link Key is a key of this type- Distributed Security Global Link Key is a key of this type
Unique link key	Optional key used to encrypt communications between a pair of nodes - may be one of the below categories:
Pre-configu unique link key	- Used between the Trust Centre and one other node- Pre-configured in Trust Centre and relevant node- Also used in joining to encrypt network key transported from Trust Centre to joining node- Install Code-derived Pre-configured Link Key is a key of this type
Trust Centre Link Key (TCLK)	- Used between the Trust Centre and one other node- Randomly generated by the Trust Centre- Distributed to node encrypted with network key and pre- configured link key (if any)- Replaces pre-configured link key (if any) but application must retain the pre-configured key in case it needs to be reinstated
Application link key	- Used between a pair of nodes, not including the Trust Centre- Randomly generated by the Trust Centre- Distributed to each node encrypted with network key and pre-configured link key (if any)

Note:

1. A pre-configured unique link key for a node can be derived from an install code on the Trust Centre using the`zps_eAplZdoAddReplaceInstallCodes()`function. Install codes are described in the ZigBee 3.0 Devices User Guide (JNUG3131).
2. In order to use a pre-configured link key in a ZigBee 3.0 application that uses the ZigBee Base Device (see Section 3.4.4, “Device types”), the Base Device attribute `bbdbJoinUsesInstallCodeKey` must be enabled and set to `TRUE`. For more information, refer to the ZigBee 3.0 Devices User Guide (JNUG3131).

Parent topic: Implementing ZigBee security

Setting up ZigBee security This section describes how to set up ZigBee security in your application code. Note that if security is enabled in a ZigBee network then network-level security is always used, while application-level security is optional.

Security is enabled on a node via the device parameter *Security Enabled*in the ZPS Configuration Editor. Enabling security also enables many-to-one routing toward the Trust Centre, which becomes a network concentrator (see Section 3.5.3).

A Trust Centre must be nominated (see Section 2.8) using the ZPS Configuration Editor. The Coordinator is normally chosen as the Trust Centre. The maximum number of nodes that will require the services of the Trust Centre must be set on the nominated node using the network parameter *Route Record Table Size*in the ZPS Configuration Editor (the default number is 4).

Security can be set up in the application code using the function **zps_vAplSecSetInitialSecurityState()**, which must be called before **zps_eAplAfInit()** and **zps_eAplZdoStartStack()** - see Section 6.1.

Note: As an alternative to using the function **zps_vAplSecSetInitialSecurityState()** in the application code, ZigBee security can be set up in the ZPS Configuration Editor (see Section 6.8.3.1).

Once **zps_vAplSecSetInitialSecurityState()** has been called and the stack has been started, the stack will automatically manage the subsequent network-level security set-up and implementation.

Network-level security set-up and application-level security set-up are further described in Section 6.8.3.1 and Section 6.8.3.2 respectively.

Note: Certain functionality on the Trust Centre can be disabled using the **zps_vSetTCLockDownOverride()** function. For more information, refer to the function description.

Network-level security set-up The function **zps_vAplSecSetInitialSecurityState()**, described above, initiates the set-up process for network-level security and requires the type of initial security key to be specified as one of:

- Pre-configured global link key
- Pre-configured unique link key

These keys are described in Section 5.8.2. They are used to encrypt the network key when it is transported to a joining node.

The Trust Centre and other nodes must be pre-programmed with the relevant pre- configured link key(s). This key can be specified in the application code for the node and referenced by **zps_vAplSecSetInitialSecurityState()** or can be set through the Key Descriptor parameter *Key*in the ZPS Configuration Editor on both the Trust Centre and other node(s). In the case of a unique link key, the IEEE/MAC address of the node must also be pre-programmed into the Trust Centre along with the link key. For the Key Descriptor parameters, refer to Section 11.7.9.

Note: Pre-configured link keys entered via the ZPS Configuration Editor are held in a Key Descriptor Table on the Trust Centre, with one entry for each node/key. The key for a node with a given IEEE/MAC address can be obtained (locally) from this table using the function **zps_psGetActiveKey()**.

The Trust Centre generates a random network key to be used in network-level communications between all nodes. When a new node joins the network, the Trust Centre transports this network key, encrypted using the appropriate pre-configured link key, to the newly joined node.

Note:

1. The application on the Trust Centre can take control (from the stack) of whether a node is allowed to join the network (possibly using its pre-configured link key) through a user-defined callback function. If required, this callback function must be registered using the **function zps_vTCSetCallback()**. For more details, refer to the function description.

2. When a device joins a ZigBee network and requires authentication which involves transporting a network key to it, the parent opens an authentication interval during which the joining device must announce itself to the network. This interval begins from the transmission of a rejoin response (if the device joins through a NWK layer rejoin) or an association response (if it joins through an IEEE 802.15.4 association). If the device fails to announce itself during this interval, the parent removes the Neighbor table entry for the joining device to ensure that the child capacity of the parent is maintained. This authentication interval must be set on all potential parent nodes via the network parameter APS Security Timeout Period (see Section 11.7), which is 1 second by default but 6 seconds is a more reasonable setting.

Parent topic: Setting up ZigBee security

Application-level security set-up Once network-level security has been set up (as described in Section 6.8.3.1), application-level security can be set up, if required. In application-level security, the communications between two nodes are encrypted/decrypted using a link key which may be global or unique:

- Global link key: This is shared between all nodes on the network and is pre- configured in all the nodes. Frame counters are not checked for freshness when using a global link key.
- Unique link key: This is exclusive to a pair of nodes that need to communicate privately. It may be a pre-configured unique link key, Trust Centre Link Key (TCLK) or application link key. Frame counters are checked for freshness to prevent rogue nodes replaying stale messages. This provides the most secure method of application security.

The different types of link key are described in Section 6.8.2 and summarized in Table 4.

In order to set up application-level security with a unique link key between two nodes, the function zps_eAplZdoRequestKeyReq() must be called on one of the nodes to request a link key from the Trust Centre. There are two possibilities:

- To request a Trust Centre Link Key (TCLK) for communication between the local node and the Trust Centre - the Trust Centre will respond with the requested link key
- To request an application link key for communication with another node that is not the Trust Centre (in this case, the IEEE/MAC address of the other node must be supplied in the function call) - the Trust Centre will send the requested link key to both nodes

The Trust Centre will ignore the request if the node is not permitted to send APS secured data. The Trust Centre responses are encrypted as follows:

- If a link key exists for communications between the Trust Centre and the target node (for example, a pre-configured link key), this key and the network key are both used to encrypt the requested link key.
- Otherwise, only the network key is used to encrypt the requested link key.

On receiving the link key, the ZigBee stack on the node will automatically save the key. The event zps_EVENT_ZDO_LINK_KEY is generated to indicate that the link key is available. Any subsequent unicast or bound data transfer between these two nodes can opt to use this key (zps_E_APL_AF_SECURE_APL).

Note: An application link key can be introduced directly by the application using the function **zps_eAplZdoAddReplaceLinkKey()**.

Note:

1. When a link key is used to encrypt a data packet, the packet payload is encrypted at the application level using the link key and then the packet is encrypted at the ZigBee stack NWK layer using the network key (therefore, both keys are used).
2. When application-level security is used in sending data, the IEEE/MAC address and network address of the target node must be available through the local Address Map table - see Section 6.2.3.

Parent topic: Setting up ZigBee security

Parent topic: Implementing ZigBee security

Security key modification The network key and an application link key can be changed while the network is operating, as described below.

Network key modification It is possible to store more than one network key on a node, although only one key can be active at any one time. Each network key is identified by means of a unique 'key sequence number' assigned by the Trust Centre application.

A new network key can be installed in a node in one of two ways:

- Distributed by the Trust Centre to one or multiple nodes of the network using the function **zps_eAplZdoTransportNwkKey()**, which requires the associated key sequence number to be specified
- Requested from the Trust Centre by calling the function

zps_eAplZdoRequestKeyReq() on the node that needs the network key

On reaching its destination(s), the transported key is automatically saved but not activated. A stored network key can be adopted as the active key using the function **zps_eAplZdoSwitchKeyReq()**, which is called on the Trust Centre and which identifies the required key by means of its unique sequence number.

Parent topic: Security key modification

Application link key modification An application link key can be introduced or replaced by the application using **zps_eAplZdoAddReplaceLinkKey()**. If a link key already exists for the same node- pair, it will be replaced by the new link key. The function must be called on both nodes in the pair.

Parent topic: Security key modification

Parent topic: Implementing ZigBee security

Parent topic:*Application coding with ZigBee PRO APIs*

Using support software features This section describes certain support software features and how to include them in your application code:

- Message queues are described in Section 6.9.1
- Software timers are described in Section 6.9.2 The referenced API resources are detailed in *Chapter 10*.

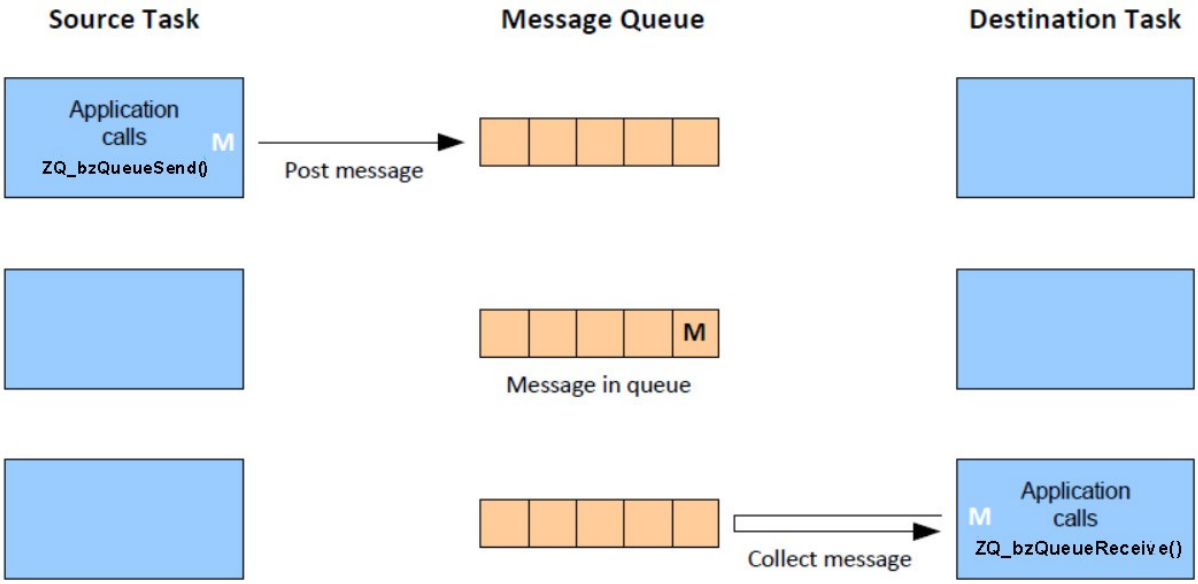
Message queues Communications between application tasks on a node are implemented via message queues. The application can create a dedicated message queue for a particular communication channel. A set of functions are provided to implement message queues, as indicated in Section 6.9.1.1below (these functions are detailed in Section10.1). The stack requires certain standard queues, as indicated in Section 6.9.1.2 below.

Note: To allow the device to enter sleep mode, the message queues must not contain any messages. All message queues must first be emptied.

General queue management A queue can be created using the function **ZQ_vZQueueCreate()**. This function allows the queue size (number of messages that it can hold) and the size of a message to be specified. A queue is given a unique handle, which is a pointer to a tszQueuestructure containing up-to-date information about the queue (see Section101.2.1).

A message can be placed in a (created) queue using the function **ZQ_bZQueueSend()** and a message can be retrieved from a queue using the function **ZQ_bZQueueReceive()**. This is illustrated in the following figure.

Sending/Receiving a Message via a Message Queue



When the above two functions are called, the tszQueuestructure for the queue is automatically updated to reflect the new state of the queue. Retrieving a message results in the message being deleted from the queue. The application must regularly poll a message queue through which it expects to receive messages. It can do this by periodically calling the **ZQ_bQueueIsEmpty()** function, which checks whether the queue is empty. If the queue is not empty, it should call **ZQ_bZQueueReceive()** until there are no more messages in the queue. The number of messages currently waiting to be collected from the queue can be obtained using the function **ZQ_u32QueueGetQueueMessageWaiting()**.

Parent topic: Message queues

Standard stack queues Three standard queues must be created by the application for use by the stack:

- Queue with handle zps_msgMlmeDcfmInd to receive IEEE 802.15.4 MAC command packets from other nodes
- Queue with handle zps_msgMcpsDcfmInd to receive IEEE 802.15.4 MAC data packets from other nodes
- Queue with handle zps_TimeEvents to receive internal software timer events (such as a timer expiry event)

Example code for the creation of these queues is provided below:

```
ZQ_vZQueueCreate(&zps_msgMlmeDcfmInd, MLME_QUEUE_SIZE, sizeof(MAC_tsMlmeVsDcfmInd),(uint8*)asMacMlmeVsDcfmInd);
```

```
ZQ_vZQueueCreate(&zps_msgMcpsDcfmInd, MCPS_QUEUE_SIZE, sizeof(MAC_tsMcpsVsDcfmInd), (uint8*)asMacMcpsDcfmInd);
```

```
ZQ_vQueueCreate(&zps_TimeEvents,TIMER_QUEUE_SIZE,sizeof(zps_tsTimeEvent),
(uint8*)asTimeEvent);
```

You simply need to include the above code in your application. You do not need to process these queues in your code.

More information on the receive queues is provided in Received message queues.

Parent topic: Message queues

Parent topic: Using support software features

Software timers The ZigBee 3.0 SDK provides resources that allow an application to implement and interact with software timers on the local node. Multiple software timers can be used concurrently and they are all derived from the same source counter, which is the ZigBee Tick Timer.

Note: To allow the device to enter sleep mode, no software timers should be active. Any running software timers must first be stopped and all timers must be closed.

Setting up timers To set up software timers in your application code, you must:

- Declare an array of `ZTIMER_tsTimer` structures (see Section 10.2.2.1), where each element/structure contains information on one timer
- Call the function **ZTIMER_vTask()** in the while loop of your application - this allows the stack software to automatically update the structure for each timer as the timer runs

For each timer, a user-defined callback function must be provided, which is referenced from the timer's structure. This callback function, **ZTIMER_tpfCallback()**, is called when the timer expires (reaches its timed period) in order to perform any operations that the application requires as a result of the timer expiration.

Before any of the software timers can be used, they must be collectively initialized by calling the function **ZTIMER_eInit()**. This function takes the array of timer structures as an input.

Before an individual timer can be used, it must be opened using the function **ZTIMER_eOpen()**. Similarly, when the timer is no longer required, it should be closed using the function **ZTIMER_eClose()**. A timer is specified in these functions by means of its index in the array of timer structures.

Parent topic: Software timers

Operating timers Once an individual software timer has been opened, it can be run one or more times before it is closed. A timer can be run by calling the function **ZTIMER_eStart()**. The timed period must be specified in milliseconds. On expiration of the timer, the user-defined callback function **ZTIMER_tpfCallback()** is called to perform any operations required by the application.

A running timer can be stopped before it expires by calling the function **ZTIMER_eStop()**. The status of an individual timer can be obtained at any time using the function **ZTIMER_eGetState()**. The possible reported states are Running, Stopped, Expired and Closed.

Parent topic: Software timers

Parent topic: Using support software features

Critical sections and Mutual Exclusion (Mutex) The ZigBee 3.0 stack software provides features to prevent sections of application code from being preempted and/or re-entered. For example, when the application is writing data to memory, it may not be desirable for this operation to be interrupted and for an interrupt service routine to start writing to the same memory block.

Two features are provided to protect sections of application code:

- **Critical Section:** A section of application code can be designated as a 'critical section', which means that the execution of this code section cannot be preempted by an interrupt with a priority level less than 12. A critical section should be short in order to avoid suspending interrupts for a long period of time.
- **Mutual Exclusion (Mutex):** It may be desirable for a section of code not to be re-entrant. A 'mutex' can be associated with a code section to prevent it from being entered again before the current execution of the section has completed.

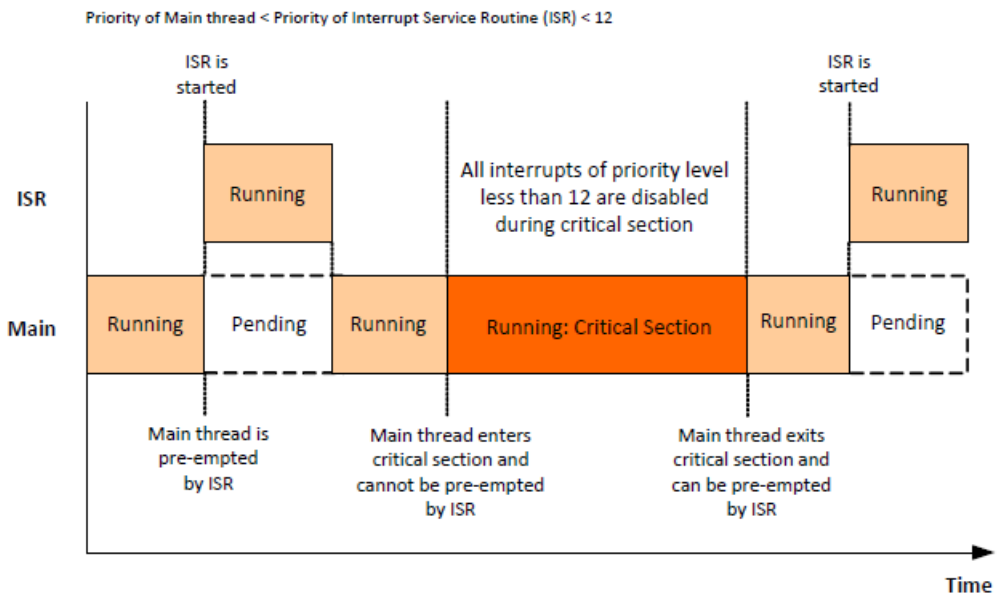
These features are described in more detail in the sub-sections below. The API resources to implement these features are detailed in Section 9.3.

Implementing a critical section

Interrupts with a priority level less than 12 cannot preempt the execution of a critical section of application code (though higher-priority interrupts can always preempt a critical section). This is illustrated in Figure 14 below, which shows the interplay between the main application thread and an interrupt service routine (ISR).

Priority of Main thread < Priority of Interrupt Service Routine (ISR) < 12

Critical Section Illustration



Critical section illustration

Time

A critical section of code must be delimited by the following two functions:

- **zps_eEnterCriticalSection()** must be called at the start of the critical section.
- **zps_eExitCriticalSection()** must be called at the end of the critical section.

A mutex can also be optionally associated with a critical section, to protect the section from re-entrancy. If required, the mutex can be specified in a parameter of **zps_eEnterCriticalSection()**. Mutexes are described in Section 5.9.3.2.

To implement critical sections, the application must maintain a 'priority level' value u8Level(see Section 9.3.2.1) which contains the current priority level of the main application thread (when critical sections are not being executed). When a critical section is entered, the priority level of the main thread is increased such that interrupts with a priority of 11 or less cannot preempt the main thread. At the end of the critical section, the priority level of the main thread is returned to the value that was contained in u8Levelbefore the critical section was entered.

Parent topic: Implementing a critical section

Parent topic: Critical sections and Mutual Exclusion (Mutex)

Implementing a Mutex A mutex can be associated with a section of application code to prevent the section from being re-entered before the current execution of the section has finished. The section of code to which the mutex will be applied must be delimited by the following two functions:

- **zps_u8GrabMutexLock()** must be called at the start of the code section.
- **zps_u8ReleaseMutexLock()** must be called at the end of the code section.

It is also possible to apply a mutex to a critical section, as described in Section 5.9.3.1.

When applying a mutex, a pointer must be provided to a user-defined mutex function with the following prototype:

((bool_t*) (*) (void))

This function must define and maintain a Boolean flag which indicates whether the corresponding mutex is active (TRUE) or inactive (FALSE). This flag is used by the API functions to determine whether the specified mutex is available. If this flag reads as FALSE at the start of the relevant code section, the mutex is applied and the above mutex function must set the flag to TRUE, but if the flag is already TRUE then the mutex cannot be applied (and the API function returns with a failure).

To implement mutex protection, the application must maintain a 'priority level' value u8Level(see Section 9.3.2.1) which contains the current priority level of the main application thread (when mutex-protected sections are not being executed). When a mutex is applied, the priority level of the main thread is increased such that interrupts with a priority of 11 or less cannot preempt the main thread. When the mutex is released, the priority level of the main thread is returned to the value that was contained in u8Levelbefore the mutex was applied.

Parent topic: Critical sections and Mutual Exclusion (Mutex)

Parent topic: Using support software features

Parent topic:*Application coding with ZigBee PRO APIs*

Advanced features This section describes the implementation of advanced ZigBee features that have been introduced in ZigBee 3.0.

End device aging A Router that is a parent needs to maintain its Neighbor table. This involves discarding inactive children (that may have left the network) in order to make way for potential new children. An End Device Aging mechanism is available to support this maintenance.

In this mechanism, a timeout is applied to every child entry in the Router's Neighbor table. If a packet, called a 'keep-alive' packet, is not received from an End Device child before its timeout expires, the child is assumed to be no longer active and is removed from the table (and therefore from the Router's children).

Timeout period The timeout period is specific to an individual child and is set on the End Device using the function **zps_bAplAfSetEndDeviceTimeout()**. This period is communicated to the parent via an End Device Timeout Request when the End Device joins (or re-joins) the network. The timeout is applied by the Router to the Neighbor table entry for the End Device. The arrival of a keep-alive packet from the End Device will result in the timeout being re-started from the beginning. If the timeout is allowed to expire (without a keep-alive packet), the Router will delete the relevant child entry from the Neighbor table.

- Note 1: The Router initially sets the timeout for all End Device children to the default value defined in the NIB, which is 256 minutes in the NXP software. The timeout will remain at this value unless changed by the End Device, as described above.
- Note 2: After receiving the End Device Timeout Request, the parent will send an End Device Timeout Response to the End Device, indicating the outcome of the request. If the request has been successful, the End Device can subsequently send keep-alive packets

Parent topic: End device aging

Keep-alive packets A keep-alive packet can be sent from the End Device using the function **zps_eAplAfSendKeepAlive()**. It is recommended that this function is called at least three times within the timeout period defined for the End Device, in order to prevent the child from being accidentally removed from the network due to missed keep-alive packets at the parent.

A keep-alive packet can be either of the following types:

- **MAC Data Poll:** In this case, the parent may send pending data back to the End Device. The arrival of this data at the End Device will be indicated by a zps_EVENT_AF_DATA_INDICATION event (as described in Section 6.5.2).
- **End Device Timeout Request**: This packet type simply has the effect of re- starting the timeout for the End Device on the parent, which will return an End Device Timeout Response to the End Device, indicating the outcome of the request.

The keep-alive packet type to be used is determined by the Router parent and is configured in the NIB on the parent - in the NXP software, a Router is configured to accept either packet type, by default. This information is communicated to the End Device in the initial End Device Timeout Response that is sent to the End Device on joining the network. The **zps_eAplAfSendKeepAlive()** function will then automatically send the appropriate keep-alive packet type - where either packet type is accepted by the parent, the function sends a Data Poll packet.

Parent topic: End device aging

Parent topic: Advanced features

Distributed security networks In a traditional ZigBee network, security is implemented by a Trust Centre, which is normally the Coordinator - this uses a centralized security scheme. In a distributed security network, any Router node can manage security and so security management is distributed throughout the network. A distributed security network does not have a Coordinator/Trust Centre, and consists only of Routers and End Devices - any Router can create the network.

In a distributed security network, only network-level security can be implemented. A network key is generated by the Router that creates the network (as described in Section 6.8.3) and is passed on to other nodes, including other Routers, as the network grows. During this distribution, the network key is encrypted using a 'Distributed Security Global Link Key', which is a type of pre-configured global link key (see Section 6.8.2).

A distributed security network can be started on a Router node using the function **zps_eAplFormDistributedNetworkRouter()**. The start parameters are specified through a zps_tsAftsStartParamsDistributedstructure (see Section 8.2.3.7). These parameters include:

- PAN ID
- Extended PAN ID (EPID)
- Radio channel
- Pointer to a location to receive the network key

This first node of the network will generate the network key (saved to the above location) and pass this key to nodes that join it.

The function **zps_eAplFormDistributedNetworkRouter()** can also be called on other Router nodes to join them to the network. An End Device can be joined to a distributed network using the function **zps_eAplInitEndDeviceDistributed()**.

However, these nodes are more likely to be introduced to the network via other commissioning methods, such as Touchlink and NFC commissioning.

Parent topic: Advanced features

Filtering packets on LQI Value/Link cost This section describes the operation and configuration of the filtering of received data packets based on LQI value (detected signal strength). Packet filtering results in some received packets with low LQI values being discarded.

In practice, the measured LQI values of packets are translated into 'link cost' values for filtering, as detailed in Section 6.10.3.1.

Packet filtering is optional and can be beneficial during:

- network joining
- route discovery
- normal network operation

The operation and benefits of packet filtering are described in Section 6.10.3.2. Packet filtering can be enabled using the function **zps_vAplAfEnableMcpsFilter()** and modified as described in Section 6.10.3.3.

Link cost For the purpose of packet filtering, LQI values are translated into 'link cost' values. Thus, a range of LQI values maps to a single link cost, which is an integer value. The default mappings implemented by the ZigBee PRO stack are shown in the table below.

'LQI to Link Cost' Mappings

LQI Range	Link Cost
51	1
46 - 50	2
41 - 45	3
39 - 40	4
36 - 38	5
25 - 35	6
24	7

The above mappings can be modified, as described in Section 6.10.3.3. A link cost of 5 is used as the packet filtering threshold by the NXP ZigBee PRO stack. Thus, packets with link costs greater than 5 may be discarded. For the device, this threshold is more suitable than the value of 3 proposed in the ZigBee specification. However, the threshold is configurable, as described in Section 6.10.3.3.

Parent topic: Filtering packets on LQI Value/Link cost

Packet filtering in operation Packet filtering is an optional feature of the ZigBee PRO stack that is applied by the IEEE 802.15.4 MAC layer. It is useful during network joining, route discovery and normal network operation to optimize the processing of received packets.

Network joining During network joining, a form of packet filtering is applied to the results of the network discovery phase. Any potential parents that have been discovered are filtered such that nodes with link costs greater than 5 (low LQI values) are discarded. This feature aids the formation of networks with strong links between neighbors and is most effective in dense networks. For more information about this process during network joining, refer to the ZigBee specification.

Parent topic: Packet filtering in operation

Route discovery and normal network operation In a large network, traffic levels are high during both route discovery and normal operation, and a node is likely to receive many data packets. There is, however, limited storage capacity on a node to hold these packets until they can be processed. To restrict the number of received packets that are submitted to the receive queue, the following filtering system is applied:

- All unicast packets are queued (without filtering) provided that sufficient space is available in the receive queue.
- Broadcast packets are queued provided that at least 50% of the receive queue capacity is free, otherwise the packet filtering mechanism is applied and only packets with a link cost of 5 or less are queued.

During route discovery, this filtering prevents nodes with low associated LQI values from being entered into the Neighbor table, allowing reliable routes to be established. For example, it may be more desirable to establish a route comprising multiple hops with good LQI values than a single hop with a poor LQI value.

Parent topic: Packet filtering in operation

Parent topic: Filtering packets on LQI Value/Link cost

Packet filtering configuration Packet filtering is disabled by default but can be enabled and re-configured as described below.

Basic configuration The function **zps_vAplAfEnableMcpsFilter()** allows the stack's packet filtering to be enabled and the link cost threshold to be adjusted (from the default value of 5). This function is detailed in Section 8.1.1. If required, it can be called at any time after **zps_eAplAfInit()**.

Parent topic: Packet filtering configuration

Link cost configuration The mappings between LQI values and link costs can be modified from the default mappings detailed in Section 6.10.3.1. To modify the mappings, the following function must be user-defined, which translates an LQI value (input) into a link cost (output):

```
uint8 APP\_u8LinkCost\(uint8 u8Lqi\);
```

An example function that implements the default mapping is shown below:

```
PRIVATE uint8 APP_u8LinkCost ( uint8 u8Lqi )
        {
        uint8 u8Lc;
        if (u8Lqi > 50)
        {
        u8Lc = 1;
        }
        else if ((u8Lqi <= 50) && (u8Lqi > 45))
        {
```

(continues on next page)

(continued from previous page)

```
        u8Lc = 2;
        }
        else if ((u8Lqi <= 45) && (u8Lqi > 40))
        {
        u8Lc = 3;
        }
        else if ((u8Lqi <= 40) && (u8Lqi > 38))
        {
        u8Lc = 4;
        }
        else if ((u8Lqi <= 38) && (u8Lqi > 35))
        {
        u8Lc = 5;
        }
        else if ((u8Lqi <= 35) && (u8Lqi > 24))
        {
        u8Lc = 6;
        }
        else
        {
        u8Lc = 7;
        }
        return u8Lc;
        }
```

The above function must be registered as a callback function using the following callback registration function **zps_vNwkLinkCostCallbackRegister()**, which is detailed in Section 8.1.1. This function takes a pointer to the **APP_u8LinkCost()** function to be registered. If required, the registration function must be called before **zps_eAplAfInit()**, and on both cold and warm starts.

Parent topic: Packet filtering configuration

Parent topic: Filtering packets on LQI Value/Link cost

Parent topic: Advanced features

Device permissions The function **zps_eAplZdoSetDevicePermission()** allows certain permissions to be set on the local device. These permissions are as follows:

Device permissions

Enumeration	Description
zps_DEVICE_PERMISSIONS_ALL_PERM	Allow all requests from other nodes
zps_DEVICE_PERMISSIONS_JOIN_DISA	Do not allow join requests from other nodes
zps_DEVICE_PERMISSIONS_DATA_REQ	Do not allow data requests from other nodes and disable end-to-end acknowledgments

When a device joins the network, the ALL_PERMITED option is set by default, so the device can respond to requests from other nodes.

However, if the network employs security set up using the ZigBee Key Establishment cluster (for example, a Smart Energy network), it is necessary to disallow data requests and end-to-end acknowledgments on the newly joined node during the key establishment process. The application must do this as follows:

1. Once an event has occurred to indicate that the device has joined the network (the event zps_EVENT_NWK_JOINED_AS_ROUTER or zps_EVENT_NWK_JOINED_AS_ENDDEVICE), the application must disallow data requests and APS end-to-end acknowledgments by calling **zps_eAplZdoSetDevicePermission()** with the option DATA_REQUEST_DISALLOWED.

2. The key establishment process can then be started using the function provided for the Key Establishment cluster.
3. Once the key establishment process has successfully completed, data requests and APS end-to-end acknowledgments can be allowed again by calling **zps_eAplZdoSetDevicePermission()** with the ALL_PERMITED option.

The key establishment process and associated resources are fully described in the documentation for the Key Establishment cluster (for example, in the *ZigBee Smart Energy User Guide)*.

Parent topic: Advanced features

Parent topic:*Application coding with ZigBee PRO APIs*

ZigBee Device Objects (ZDO) API The chapter describes the resources of the ZigBee Device Objects (ZDO) API. This API is primarily concerned with starting, forming, and modifying a ZigBee PRO network. The API is defined in the header file **zps_apl_zdo.h**.

In this chapter:

- ZDO API functions details the ZDO API functions.
- ZDO enumerations details the ZDO API enumerations.

ZDO API functions The ZDO API functions are divided into the following categories:

1. **Network Deployment**functions, described in Network deployment functions.
2. **Security**functions, described in Security functions.
3. **Addressing**functions, described in Addressing functions.
4. **Routing**functions, described in Routing functions.
5. **Object Handle**functions, described in Object Handle functions.
6. **Optional Cluster**function, described in Optional Cluster function.

Network deployment functions The ZDO Network Deployment functions are used to start the ZigBee PRO stack, and allow devices to join the network and bind to each other, as well as leave the network.

The functions are listed below.

Function page

1. ZPS_eAplZdoStartStack
2. ZPS_vDefaultStack
3. ZPS_eAplZdoGetDeviceType
4. ZPS_eAplZdoDiscoverNetworks
5. ZPS_eAplZdoJoinNetwork
6. ZPS_eAplZdoRejoinNetwork
7. ZPS_eAplZdoDirectJoinNetwork
8. ZPS_eAplZdoOrphanRejoinNetwork
9. ZPS_eAplZdoPermitJoining
10. ZPS_u16AplZdoGetNetworkPanId
11. ZPS_u64AplZdoGetNetworkExtendedPanId

12. ZPS_u8AplZdoGetRadioChannel
13. ZPS_eAplZdoBind
14. ZPS_eAplZdoUnbind
15. ZPS_eAplZdoBindGroup
16. ZPS_eAplZdoUnbindGroup
17. ZPS_ePurgeBindTable
18. ZPS_eAplZdoPoll
19. ZPS_eAplZdoLeaveNetwork
20. ZPS_vNwkNibSetLeaveAllowed
21. ZPS_vNwkNibSetLeaveRejoin
22. ZPS_vSetTablesClearOnLeaveWithoutRejoin
23. ZPS_vNtSetUsedStatus
24. ZPS_vNwkSendNwkStatusCommand
25. ZPS_eAplZdoRegisterZdoLeaveActionCallback

Note: The ZDO initialization and start stack functions use network parameter values that have been pre-set and saved using the steps described in Chapter 13, *ZPS Configuration Editor*.

Parent topic: Network deployment functions

ZPS_eAplZdoStartStack

```
ZPS\__teStatus ZPS\__eAplZdoStartStack\(void\);
```

Description

This function starts the ZigBee PRO stack. The steps taken depend on the node type:

- If the device is the Coordinator, this function starts the network formation process.
- If the device is a Router or End Device, this function starts the network discovery process - that is, the device searches for a network to join.

When the stack starts, the 2400 MHz radio channel to be used by the device is selected. The channels (in the range 11 to 26) available to the device should be specified in advance using the ZPS Configuration Editor (see *Chapter 13*) and can be either of the following:

- A fixed channel
- A set of channels for a channel scan:
  - If the device is the Coordinator, this is the set of channels that the device scans to find a suitable operating channel for the network.
  - If the device is a Router or End Device, this is the set of channels that the device scans to find a network to join.

If this function successfully initiates network formation or discovery, it returns ZPS_E_SUCCESS. Subsequent results from this process are then reported through stack events (see Events for details of these events):

- If the Coordinator successfully creates a network, the event ZPS_EVENT_NWK_STARTED is generated. Otherwise, the event ZPS_EVENT_NWK_FAILED_TO_START is generated.
- When the network discovery process for a Router or End Device is complete, the subsequent actions depend on the Extended PAN ID (EPID) that is pre-set using the ZPS Configuration Editor:

- If a zero EPID value was pre-set, the stack event ZPS_EVENT_NWK_DISCOVERY_COMPLETE is generated. This includes a list of the detected networks and the index (in the list) of the recommended network to join. You can then call **ZPS_eAplZdoJoinNetwork()** to join the desired network.
- If a non-zero EPID value was pre-set, the device automatically attempts to join the network with this EPID, provided that such a network has been discovered. Note that the 'permit joining' setting of the potential parent is ignored.

The maximum depth (number of levels below the Coordinator) of the network is 15.

Parameters None

Parent topic:ZPS_eAplZdoStartStack

Returns

- ZPS_E_SUCCESS (stack started and network formation/discovery begun)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoStartStack

Parent topic: Network deployment functions

ZPS_vDefaultStack

```
void ZPS\__vDefaultStack\(void\);
```

Description

This function can be used to reset the ZigBee PRO stack to its default state. It removes previous context data for the stack, but leaves NWK layer frame counters intact.

Note: After calling this function, all security keys must be re-configured.

Parameters None

Returns None

Parent topic: Network deployment functions

ZPS_eAplZdoGetDeviceType

```
ZPS\__teZdoDeviceType ZPS\__eAplZdoGetDeviceType\(void\);
```

Description

This function can be used to obtain the ZigBee node type (Coordinator, Router, or End Device) of the local node.

Parameters

None

Returns

ZigBee node type, one of:

- ZPS_ZDO_DEVICE_COORD (Coordinator)
- ZPS_ZDO_DEVICE_ROUTER (Router)

- ZPS_ZDO_DEVICE_ENDDEVICE (End Device)

Parent topic: Network deployment functions

ZPS_eAplZdoDiscoverNetworks

```
ZPS\_teStatus ZPS\_eAplZdoDiscoverNetworks\(uint32 u32ChannelMask\);
```

Description

This function can be used by a Router or End Device to initiate a network discovery

- that is, to find a network to join.

A network discovery is performed when the stack is started using the function **ZPS_eAplZdoStartStack()**. The function **ZPS_eAplZdoDiscoverNetworks()** can be used to perform subsequent network discoveries (for example, if the initial search did not yield any suitable networks).

As part of this function call, you must specify a value which indicates the 2400-MHz radio channels (numbered 11 to 26) to be used in the network search. There are two ways of setting this parameter:

- A single value in the range 11 to 26 can be specified, indicating that the corresponding channel (and no other) must be used - for example, 12 indicates use channel 12.
- A 32-bit mask can be used to specify a set of channels that the device will scan to find a network - each of bits 11 to 26 represents the corresponding radio channel, where the channel will be included in the scan if the bit is set to 1 (and excluded if cleared to 0). Therefore, the value 0x07FFF800 represents all channels.

Note: If an invalid value is specified for this parameter, the default value of 0x07FFF800 (all channels) will be used.

If this function successfully initiates a network discovery, ZPS_E_SUCCESS will be returned. The network discovery results will then be reported through the event ZPS_EVENT_NWK_DISCOVERY_COMPLETE (for details of this event, refer to Section 7.2.2.9). This includes a list of the detected networks and the index (in the list) of the recommended network to join. You should then call **ZPS_eAplZdoJoinNetwork()** to join the desired network.

Parameters

u32ChannelMask Radio channel(s) for network discovery (see above)

Returns

ZPS_E_SUCCESS (network discovery started)

- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic: Network deployment functions

ZPS_eAplZdoJoinNetwork

```
ZPS\_teStatus ZPS\_eAplZdoJoinNetwork\(ZPS\_tsNwkNetworkDescr \*psNetworkDescr\);
```

Description

This function can be used by a Router or End Device to send a request to join a particular network, following a network discovery.

The required network is specified using its network descriptor, obtained in a ZPS_EVENT_NWK_DISCOVERY_COMPLETE event which results from a

network discovery previously implemented using **ZPS_eAplZdoStartStack()** or **ZPS_eAplZdoDiscoverNetworks()**. For details of this event, refer to ZPS_tsAfNwkDiscoveryEvent.

If the join request is successfully sent, the function will return ZPS_E_SUCCESS (note that this does not mean that device has joined the network). The result of the join request will then be reported through a stack event (see Events for details of these events):

- If the device successfully joined the network as a Router, the event **ZPS_EVENT_NWK_JOINED_AS_ROUTER** is generated. The allocated 16-bit network address of the Router is returned as part of this stack event.
- If the device successfully joined the network as an End Device, the event **ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE** is generated. The allocated 16-bit network address of the End Device is returned as part of this stack event.
- If the join request was unsuccessful, the event ZPS_EVENT_NWK_FAILED_TO_JOIN is generated.

Note: Note that nodes can join a ZigBee PRO network to a maximum depth of 15 (levels below the Coordinator).

Parameters

**psNetworkDescr* Pointer to network descriptor of network to join.

Returns

- ZPS_E_SUCCESS (join request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic: Network deployment functions

ZPS_eAplZdoRejoinNetwork

```
**ZPS\__teStatus ZPS\__eAplZdoRejoinNetwork\(bool\__t bWithDiscovery\);**
```

This function can be used by an active Router or End Device to send a request to rejoin its previous network. The function should be called if the application detects that it has lost its connection to the network - this is indicated by an excessive number of failed communications (for example, with many missing acknowledgments).

Options are provided to first perform a network discovery to find potential parents to join or simply rejoin the previous parent.

If the rejoin request is successfully sent, the function returns ZPS_E_SUCCESS. Note that this does not mean that device has rejoined the network. The result of the rejoin request is then reported through a stack event (see Section 10.1 for details of these events):

- If the device successfully rejoined the network as a Router, the event ZPS_EVENT_NWK_JOINED_AS_ROUTER is generated.
- If the device successfully rejoined the network as an End Device, the event ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE is generated.
- If the rejoin request was unsuccessful, the event ZPS_EVENT_NWK_FAILED_TO_JOIN is generated.

In the case of a successful rejoin, the node will retain its previously allocated 16-bit network address.

Note that the 'permit joining' status of the potential parent is ignored during a rejoin.

Parameters

bWithDiscovery Specifies whether a network discovery is required:

- TRUE - perform network discovery before rejoining
- FALSE - rejoin previous parent
- TRUE - perform network discovery before rejoining
- FALSE - rejoin previous parent

ZPS_E_SUCCESS (rejoin request successfully sent)

- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic: Network deployment functions

ZPS_eAplZdoDirectJoinNetwork

```
ZPS_teStatus ZPS_eAplZdoDirectJoinNetwork( uint64 u64Addr, uint16 u16Addr, uint8 u8Capability);
```

Description

This function can be used on a Router and on the Coordinator to pre-determine the child nodes that will directly join it. The function is called to register each child node separately, and the IEEE/MAC and network addresses of the child node must be specified. The function adds the registered node to its Neighbor table (it actually adds the node's IEEE/MAC address to the MAC Address table and then includes the index of this address in a Neighbor table entry for the node).

The function must be called only when the parent node is fully up and running in the network. Since the child node has not yet joined the network but is in the Neighbor table, it will be perceived by the parent as having been orphaned. Therefore, when the child node attempts to join the network, it must perform a rejoin as an orphan by calling the function ZPS_eAplZdoOrphanRejoinNetwork\(\).

Note: You should only modify the Neighbor table using this function and never write to it directly.

Parameters

- *u64Addr* IEEE/MAC address of child node to be registered
- *u16Addr* Network address of child node to be registered
- *u8Capability* A bitmap indicating the operational capabilities of the child node - this bitmap is detailed in Table in section ZPS_tsAfNwkJoinIndEvent.

Returns

- ZPS_E_SUCCESS (child node successfully registered)
- ZPS_APL_APS_E_ILLEGAL_REQUEST (address 0x0, address 0xFFFFFFFFFFFFFFFF, own address, ZDO busy)
- ZPS_NWK_ENUM_ALREADY_PRESENT
- ZPS_NWK_ENUM_NEIGHBOR_TABLE_FULL

Parent topic: Network deployment functions

ZPS_eAplZdoOrphanRejoinNetwork

```
ZPS_teStatus ZPS_eAplZdoOrphanRejoinNetwork(void);
```

This function can be used by an orphaned node to attempt to rejoin the network - the orphaned node may be an End Device or a Router. The function should also be used for a first-time join for which the parent has been pre-determined using the function **ZPS_eAplZdoDirectJoinNetwork()**.

The function starts the stack on the node. Therefore, when this function is used, there is no need to explicitly start the stack using **ZPS_eAplZdoStartStack()**.

If the rejoin request is successfully sent, the function will return ZPS_E_SUCCESS (note that this does not mean that device has rejoined the network). The result of the rejoin request will then be reported through a stack event (see Section 10.1 for details of these events):

- If the device successfully rejoined the network as a Router, the event ZPS_EVENT_NWK_JOINED_AS_ROUTER is generated.
- If the device successfully rejoined the network as an End Device, the event ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE is generated.
- If the rejoin request was unsuccessful, the event ZPS_EVENT_NWK_FAILED_TO_JOIN is generated.

In the case of a successful rejoin of a genuinely orphaned node, the node will retain its previously allocated 16-bit network address.

Note: The 'permit joining' status of the potential parent is ignored during a rejoin.

Parameters None

Parent topic:ZPS_eAplZdoOrphanRejoinNetwork

Returns

- ZPS_E_SUCCESS (rejoin request successfully sent)
- ZPS_APL_APS_E_ILLEGAL_REQUEST (missing EPID, called from Coordinator, ZDO busy)

Parent topic:ZPS_eAplZdoOrphanRejoinNetwork

Parent topic: Network deployment functions

ZPS_eAplZdoPermitJoining

```
ZPS_teStatus ZPS_eAplZdoPermitJoining(
                        uint8 u8PermitDuration);
```

Description This function can be used on a Router or the Coordinator to control whether new child nodes are allowed to join it - that is, to set the node's 'permit joining' status. The function can be used to enable joining permanently or for a fixed duration, or to disable joining (permanently).

The specified parameter value determines the 'permit joining' status, as follows:

- 0: Disables joining
- 1- 254: Enables joining for specified time interval, in seconds
- 255: Enables joining permanently

For example, if the parameter is set to 60, joining is enabled for the next 60 seconds and then automatically disabled.

Note: The 'permit joining' setting of a device is ignored during a join attempt in which a non-zero Extended PAN ID is specified on the joining device and during any rejoin attempt.

Parent topic:ZPS_eAplZdoPermitJoining

Parameters *u8PermitDuration* Time duration, in seconds, for which joining will be permitted (see above)

Parent topic:ZPS_eAplZdoPermitJoining

Returns

- ZPS_E_SUCCESS ('permit joining' status successfully set)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoPermitJoining

Parent topic: Network deployment functions

ZPS_u16AplZdoGetNetworkPanId

```
uint16 ZPS_u16AplZdoGetNetworkPanId(void);
```

Description This function obtains the 16-bit PAN ID of the ZigBee network to which the local node currently belongs.

Parent topic:ZPS_u16AplZdoGetNetworkPanId

Parameters None.

Parent topic:ZPS_u16AplZdoGetNetworkPanId

Returns PAN ID of current network.

Parent topic:ZPS_u16AplZdoGetNetworkPanId

Parent topic: Network deployment functions

ZPS_u64AplZdoGetNetworkExtendedPanId

```
uint64 ZPS_u64AplZdoGetNetworkExtendedPanId(void)
```

Description This function obtains the 64-bit Extended PAN ID (EPID) of the ZigBee PRO network to which the local node currently belongs.

Parent topic:ZPS_u64AplZdoGetNetworkExtendedPanId

Parameters None

Parent topic:ZPS_u64AplZdoGetNetworkExtendedPanId

Returns Extended PAN ID of current network.

Parent topic:ZPS_u64AplZdoGetNetworkExtendedPanId

Parent topic: Network deployment functions

ZPS_u8AplZdoGetRadioChannel

```
uint8 ZPS_u8AplZdoGetRadioChannel(void);
```

Description This function obtains the 2400-MHz band channel in which the local node is currently operating. The channel is represented by an integer in the range 11 to 26.

Parent topic:ZPS_u8AplZdoGetRadioChannel

Parameters None.

Parent topic:ZPS_u8AplZdoGetRadioChannel

Returns Radio channel number (in range 11-26).

Parent topic:ZPS_u8AplZdoGetRadioChannel

Parent topic: Network deployment functions

ZPS_eAplZdoBind

```
ZPS_teStatus ZPS_eAplZdoBind(
                    uint16 u16ClusterId,
                    uint8 u8SrcEndpoint,
                    uint16 u16DstAddr,
                    uint64 u64DstIeeeAddr,
                    uint8 u8DstEndpoint);
```

Description This function requests a binding to be created between an endpoint on the local node and an endpoint on a remote node. The source endpoint and cluster must be specified, as well as the destination node and endpoint. The destination node is specified using both its 64-bit IEEE (MAC) address and its 16-bit network address.

The binding is added to the binding table on the local node.

A binding to multiple remote endpoints (collected into a group) can be created using the function **ZPS_eAplZdoBindGroup()**.

Parent topic:ZPS_eAplZdoBind

Parameters *u16ClusterId* Identifier of cluster on source node to be bound *u8SrcEndpoint* Number of endpoint (1-240) on source node to be bound *u16DstAddr* 16-bit network address of destination for binding *u64DstIeeeAddr* 64-bit IEEE (MAC) address of destination for binding *u8DstEndpoint* Number of endpoint on destination node to be bound

Parent topic:ZPS_eAplZdoBind

Returns

- ZPS_E_SUCCESS (binding successfully created)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoBind

Parent topic: Network deployment functions

ZPS_eAplZdoUnbind

```
ZPS_teStatus ZPS_eAplZdoUnbind(
                    uint16 u16ClusterId,
                    uint8 u8SrcEndpoint,
                    uint16 u16DstAddr,
                    uint64 u64DstIeeeAddr,
                    uint8 u8DstEndpoint);
```

Description This function requests an existing binding to be removed between an endpoint on the local node and an endpoint on a remote node, where this binding was created using the function **ZPS_eAplZdoBind()**. The source endpoint and cluster must be specified, as well as the destination node and endpoint. The destination node is specified using both its 64-bit IEEE (MAC) address and its 16-bit network address.

The binding is removed from the binding table on the local node.

Parent topic:ZPS_eAplZdoUnbind

Parameters *u16ClusterId* Identifier of bound cluster on source node *u8SrcEndpoint* Number of bound endpoint (1-240) on source node *u16DstAddr* 16-bit network address of destination for binding *u64DstIeeeAddr* 64-bit IEEE (MAC) address of destination for binding *u8DstEndpoint* Number of bound endpoint on destination node

Parent topic:ZPS_eAplZdoUnbind

Returns

- ZPS_E_SUCCESS (binding successfully removed)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoUnbind

Parent topic: Network deployment functions

ZPS_eAplZdoBindGroup

```
ZPS_teStatus ZPS_eAplZdoBindGroup(
                    uint16 u16ClusterId,
                    uint8 u8SrcEndpoint,
                    uint16 u16DstGrpAddr);
```

Description This function requests a binding to be created between an endpoint on the local node and multiple endpoints on remote nodes. The source endpoint and cluster must be specified, as well as the destination nodes/endpoints for the binding, which must be specified using a 16-bit group address, previously set up using **ZPS_eAplZdoGroupEndpointAdd()**.

The binding is added to the binding table on the local node.

Parent topic:ZPS_eAplZdoBindGroup

Parameters *u16ClusterId* Identifier of cluster on source node to be bound *u8SrcEndpoint* Number of endpoint (1-240) on source node to be bound *u16DstGrpAddr* 16-bit group address of destination group for binding

Parent topic:ZPS_eAplZdoBindGroup

Returns

- ZPS_E_SUCCESS (binding successfully created)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoBindGroup

Parent topic: Network deployment functions

ZPS_eAplZdoUnbindGroup

```
ZPS_teStatus ZPS_eAplZdoUnbindGroup(
    uint16 u16ClusterId,
    uint8 u8SrcEndpoint,
    uint16 u16DstGrpAddr);
```

Description This function requests an existing binding to be removed between an endpoint on the local node and a group of endpoints on remote nodes, where this binding was created using the function **ZPS_eAplZdoBindGroup()**. The source endpoint and cluster must be specified, as well as the destination nodes/endpoints for the binding, which must be specified using a 16-bit group address.

The binding is removed from the binding table on the local node.

Parent topic:ZPS_eAplZdoUnbindGroup

Parameters *u16ClusterId* Identifier of bound cluster on source node

u8SrcEndpoint Number of bound endpoint (1-240) on source node

u16DstGrpAddr 16-bit group address of bound destination group

Parent topic:ZPS_eAplZdoUnbindGroup

Returns

- ZPS_E_SUCCESS (binding successfully removed)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3

- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoUnbindGroup

Parent topic: Network deployment functions

ZPS_ePurgeBindTable

```
ZPS_teStatus ZPS_ePurgeBindTable(void);
```

Description This function removes all bindings from the binding table on the local node.

Parent topic:ZPS_ePurgeBindTable

Parameters None

Parent topic:ZPS_ePurgeBindTable

Returns

- ZPS_E_SUCCESS (binding successfully removed)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_ePurgeBindTable

Parent topic: Network deployment functions

ZPS_eAplZdoPoll

```
ZPS_teStatus ZPS_eAplZdoPoll(void);
```

Description This function can be used by an End Device to poll its parent for pending data.

Since an End Device is able to sleep, messages addressed to the End Device are buffered by the parent for delivery when the child is ready. This function requests this buffered data and should normally be called immediately after waking from sleep.

This function call will trigger a confirmation event, ZPS_EVENT_NWK_POLL_CONFIRM, if the poll request is successfully sent to the parent. The subsequent arrival of data from the parent is indicated by a ZPS_EVENT_APS_DATA_INDICATION event. Any messages forwarded from the parent should then be collected using the function **ZQ_bZQueueReceive()**.

Parent topic:ZPS_eAplZdoPoll

Parameters None

Parent topic:ZPS_eAplZdoPoll

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoPoll

Parent topic: Network deployment functions

ZPS_eAplZdoLeaveNetwork

```
ZPS_teStatus ZPS_eAplZdoLeaveNetwork(
                    uint64 u64Addr,
                    bool bRemoveChildren,
                    bool bRejoin);
```

Description This function can be used to request a node to leave the network. The leaving node can be a child of the requesting node or can be the requesting node itself (excluding the Coordinator).

The node being asked to leave the network is specified by means of its IEEE (MAC) address (or zero, if a node is requesting itself to leave the network). You must also:

- Use the parameter *bRemoveChildren* to specify whether children of the leaving node must leave their parent - if this is the case, the leaving node will automatically call **ZPS_eAplZdoLeaveNetwork()** for each of its children. This parameter must always be set to FALSE when the function is called on an End Device (as there are no children).
- Use the parameter *bRejoin*to specify whether the leaving node must attempt to rejoin the network (probably via another parent) immediately after leaving.

Note: If you wish to move a whole network branch from under the requesting node to a different parent node, set bRemoveChildren to FALSE and bRejoin to TRUE.

If this function successfully initiates the removal of a node, ZPS_E_SUCCESS will be returned. Subsequently, when the removal is complete, the stack event ZPS_EVENT_NWK_LEAVE_CONFIRM is generated. For details of this event, refer to Section 7.2.2.12.

Parent topic:ZPS_eAplZdoLeaveNetwork

Parameters *u64Addr* 64-bit IEEE (MAC) address of node to leave network (zero value will cause requesting node to leave network)

bRemoveChildren Boolean value indicating whether children of leaving node must leave their parent:

TRUE: Children to leave FALSE: Children not to leave

bRejoin Boolean value indicating whether leaving node must attempt to rejoin network immediately after leaving:

TRUE: Rejoin network immediately FALSE: Do not rejoin network

Parent topic:ZPS_eAplZdoLeaveNetwork

Returns

- ZPS_E_SUCCESS (removal of node successfully started)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoLeaveNetwork

Parent topic: Network deployment functions

ZPS_vNwkNibSetLeaveAllowed

```
void ZPS_vNwkNibSetLeaveAllowed(void *pvNwk,
                                bool bLeave);
```

Description This function can be called on a Router or End Device to determine whether the device should leave the network on receiving a leave request. It has no effect on a Coordinator.

- If called with *bLeave* set to TRUE, the device obeys a leave request.
- If called with *bLeave* set to FALSE, the device ignores a leave request.

Parent topic:ZPS_vNwkNibSetLeaveAllowed

Parameters

- *pvNwk* Pointer to NWK layer instance
- *bLeave*Boolean value indicating whether the device leaves the network when requested or ignores leave request messages:
  - TRUE - Obey leave request messages.
  - FALSE - Ignore leave request messages.

Parent topic:ZPS_vNwkNibSetLeaveAllowed

Returns None

Parent topic:ZPS_vNwkNibSetLeaveAllowed

Parent topic: Network deployment functions

ZPS_vNwkNibSetLeaveRejoin

```
void ZPS_vNwkNibSetLeaveRejoin(void *pvNwk,
                               bool bRejoin);
```

Description This function can be called on a Router or End Device to configure the device to automatically rejoin after leaving the network, even when a 'leave without rejoin' was requested.

- If called with *bRejoin* set to TRUE, the device will rejoin following a leave.
- If called with *bRejoin* set to FALSE, the device will not rejoin following a leave.

Parent topic:ZPS_vNwkNibSetLeaveRejoin

Parameters

- *pvNwk* Pointer to NWK layer instance
- *bLeave* Boolean value indicating whether the device will rejoin the network following a leave:
  - TRUE - Rejoin the network
  - FALSE - Do not rejoin the network

Parent topic:ZPS_vNwkNibSetLeaveRejoin

Returns None

Parent topic:ZPS_vNwkNibSetLeaveRejoin

Parent topic: Network deployment functions

ZPS_vSetTablesClearOnLeaveWithoutRejoin

```
void ZPS_vSetTablesClearOnLeaveWithoutRejoin(
                                    bool_t bClear);
```

Description This function can be called on a Router or End Device to configure whether various tabulated context data must be cleared from the node when it leaves the network without the intention to rejoin.

By default, the Neighbor table, Binding table and Group table are cleared on a Router, and the network key is cleared on a Router and End Device. In addition, other devices remove the node from their Binding tables on detecting the leave request (without the rejoin flag set).

This function can be used to over-ride this behavior in order to preserve this table data. It can also be used to later reinstate the default behavior.

Parent topic:ZPS_vSetTablesClearOnLeaveWithoutRejoin

Parameters

- *bClear* Boolean value indicating whether the node should clear the table data when leaving the network without a future rejoin:
  - TRUE - Clear table data (default behavior)
  - FALSE - Do not clear table data

Parent topic:ZPS_vSetTablesClearOnLeaveWithoutRejoin

Returns None

Parent topic:ZPS_vSetTablesClearOnLeaveWithoutRejoin

Parent topic: Network deployment functions

ZPS_vNtSetUsedStatus

```
void ZPS_vNtSetUsedStatus(
            void *pvNwk,
            ZPS_tsNwkActvNtEntry *psActvNtEntry,
            bool_t bStatus);
```

Description This function can be used to set the status of a local Neighbor Table to either 'used' or 'unused':

- Setting the status of an entry to unused effectively removes the entry from the table.
- Setting the status of an entry to used effectively adds an entry to the table.

When adding an entry to the table, it is first necessary for the application to find an entry marked unused. The entry can then be populated with data and marked as used via this function.

Parent topic:ZPS_vNtSetUsedStatus

Parameters

- *pvNwk* Pointer to NWK layer instance
- *psActvNtEntry*Pointer to Neighbor Table entry to access (this must be populated with data when adding a new entry to the table)
- *bStatus* Entry status to be set:
  - TRUE - Set entry status to 'used'
  - FALSE - Set entry status to 'unused'

Parent topic:ZPS_vNtSetUsedStatus

Returns None

Parent topic:ZPS_vNtSetUsedStatus

Parent topic: Network deployment functions

ZPS_vNwkSendNwkStatusCommand

```
void ZPS_vNwkSendNwkStatusCommand(
                void *pvNwk,
                uint16 u16DstAddress,
                uint16 u16TargetAddress,
                uint8 u8CommandId,
                uint8 u8Radius);
```

Description This function can be used to send a network status command to another node. For example, it can be used by an End Device to report a routing problem (concerning a remote node) to its parent.

Parent topic:ZPS_vNwkSendNwkStatusCommand

Parameters

- *pvNwk* Pointer to NWK layer instance
- *u16DstAddress* Network address of the remote node to which the status command relates (for example, the node for which a routing problem is being reported)
- *u16TargetAddress* Network address of the node to which the status command is to be sent (for example, the parent of the local node)
- *u8CommandId* Value representing the network status command to be sent (the possible values are provided in the ZigBee PRO specification)
- *u8Radius* Maximum number of hops permitted to target node (zero value specifies that default maximum is to be used)

Parent topic:ZPS_vNwkSendNwkStatusCommand

Returns None

Parent topic:ZPS_vNwkSendNwkStatusCommand

Parent topic: Network deployment functions

ZPS_eAplZdoRegisterZdoLeaveActionCallback

```
void ZPS_eAplZdoRegisterZdoLeaveActionCallback(
                                    void *fnPtr);
```

Description This function can be used to register a user-defined callback function that will be invoked when a leave request, a management leave request or a remove device request (from a remote node, normally the Trust Centre) is received by the local node. The callback function must determine whether the request must be obeyed or ignored by the stack - this decision may depend on the originator of the request.

The prototype of the callback function is as follows:

bool_t ZPS_bPerformLeaveActionDecider(uint8 *u8Value* **,**

uint64 *u64Address*, **uint8** *u8Flags*);

where:

- *u8Value* is an enumerated value indicating the type of request - one of: ZPS_LEAVE_ORIGIN_NLME (NLME-LEAVE.request from NWK layer) ZPS_LEAVE_ORIGIN_MGMT_LEAVE (management leave request) ZPS_LEAVE_ORIGIN_REMOVE_DEVICE (remove request from remote node)
- *u64Address* is the IEEE/MAC address of the node that issued the request
- *u8Flags* is a user-defined bitmap containing flagged information

The callback function must return TRUE to allow or FALSE to disallow the requested leave.

Parent topic:ZPS_eAplZdoRegisterZdoLeaveActionCallback

Parameters *fnPtr*: Pointer to user-defined callback function to be registered.

Parent topic:ZPS_eAplZdoRegisterZdoLeaveActionCallback

Returns None

Parent topic:ZPS_eAplZdoRegisterZdoLeaveActionCallback

Parent topic: Network deployment functions

Parent topic:ZDO API functions

Security functions The ZDO Security functions are used to set up network security (at the 'standard' level), including the keys used in the encryption/decryption of network communications.

The functions are listed below.

Function page

1. ZPS_vAplSecSetInitialSecurityState
2. ZPS_eAplZdoTransportNwkKey
3. ZPS_eAplZdoSwitchKeyReq
4. ZPS_eAplZdoRequestKeyReq
5. ZPS_eAplZdoAddReplaceLinkKey
6. ZPS_eAplZdoAddReplaceInstallCodes
7. ZPS_eAplZdoRemoveLinkKey
8. ZPS_eAplZdoRemoveDeviceReq
9. ZPS_eAplZdoSetDevicePermission
10. ZPS_bAplZdoTrustCenterSetDevicePermissions
11. ZPS_bAplZdoTrustCenterGetDevicePermissions
12. ZPS_bAplZdoTrustCenterRemoveDevice
13. ZPS_vTcInitFlash
14. ZPS_vSetTCLockDownOverride
15. ZPS_psGetActiveKey
16. ZPS_vTCSetCallback

Note:

1. Before using the above functions on a node, security must be enabled on the node via the device parameter Security Enabled in the ZPS Configuration Editor (security is enabled by default).
2. Enabling security also enables many-to-one routing toward the Trust Centre, which then becomes a network concentrator. You must set the maximum number of nodes to be serviced by the Trust Centre using its network parameter Route Record Table Size in the ZPS Configuration Editor (the default number is 4).
3. Many of the security settings and keys that are set up using the above functions can alternatively be pre-configured via the ZPS Configuration Editor.

Parent topic: Security functions

ZPS_vAplSecSetInitialSecurityState

```
ZPS_teStatus ZPS_vAplSecSetInitialSecurityState(
    ZPS_teZdoNwkKeyState eState,
    uint8 *pu8Key,
    uint8 u8KeySeqNum
    ZPS_teApsLinkKeyType eKeyType);
```

Description

This function is used to configure the initial state of ZigBee security on the local node. This requires a security key to be specified that is used in setting up network-level security.

Note: Before using this function, security must be enabled on the node via the device parameter Security Enabled in the ZPS Configuration Editor.

You must provide a pointer to an initial link key of one of the following types:

- Pre-configured global link key

- Pre-configured unique link key

These key types are described in Section 6.8.2. The network key randomly generated by the Trust Centre is communicated to the local node in encrypted form using the specified link key.

Parameters

- *eState*: The state of the link key, one of:
  - ZPS_ZDO_PRECONFIGURED_LINK_KEY
  - ZPS_ZDO_ZLL_LINK_KEY
- *pu8Key*: Pointer to pre-configured link key
- *u8KeySeqNum*: Not used when specifying a link key - ignore this parameter
- *eKeyType*: Type of link key, one of:
  - ZPS_APS_UNIQUE_LINK_KEY
  - ZPS_APS_GLOBAL_LINK_KEY

Parent topic:ZPS_vAplSecSetInitialSecurityState

Returns

- ZPS_E_SUCCESS (security state successfully initialized)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_vAplSecSetInitialSecurityState

Parent topic: Security functions

ZPS_eAplZdoTransportNwkKey

```
ZPS\_teStatus ZPS\_eAplZdoTransportNwkKey\(
  uint8 u8DstAddrMode,
  ZPS\_tuAddress uDstAddress,
  uint8 au8Key[ZPS_SEC_KEY_LENGTH],
  uint8 u8KeySeqNum,
  bool bUseParent,
  uint64 u64ParentAddr);
```

Description This function can be used on the Trust Centre to send the network key to one or multiple nodes. On reaching the target node(s), the key is only stored but can be subsequently designated the active network key using the function **ZPS_eAplZdoSwitchKeyReq()**.

The target node can be specified by means of its network address or IEEE/MAC address. A broadcast to multiple nodes in the network can be achieved by specifying a special network address or IEEE/MAC address - see Section 9.3.

If the destination is a single node, it is possible to send the key to the parent of the destination node.

Note: This function also resets the frame counter on the target node(s).

Parent topic:ZPS_eAplZdoTransportNwkKey

Parameters

- *u8DstAddrMode* Type of destination address:
  - ZPS_E_ADDR_MODE_SHORT - 16-bit network address.
  - ZPS_E_ADDR_MODE_IEEE - 64-bit IEEE/MAC address.
  - All other values are reserved.
- *uDstAddress:* Destination address (address type as specified through *u8DstAddrMode*) - special broadcast addresses are detailed in Section 9.3
- *au8Key[]:* Array containing the network key to be transported. This array has a length equal to ZPS_SEC_KEY_LENGTH
- *u8KeySeqNum:* Sequence number of the specified key
- *bUseParent:* Indicates whether to send key to parent of target node:
  - TRUE - send to parent
  - FALSE - do not send to parent
- *u64ParentAddr:* 64-bit IEEE/MAC address of parent (if used).

Parent topic:ZPS_eAplZdoTransportNwkKey

Returns

- ZPS_E_SUCCESS (key successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoTransportNwkKey

Parent topic: Security functions

ZPS_eAplZdoSwitchKeyReq

```
ZPS_teStatus ZPS_eAplZdoSwitchKeyReq(
        uint8 u8DstAddrMode,
        ZPS_tuAddress uDstAddress,
        uint8 u8KeySeqNum);
```

Description This function can be used (normally by the Trust Centre) to request one or multiple nodes to switch to a different active network key. The new network key is specified using its unique sequence number and the key must have been pre-loaded into the target node(s) using the function **ZPS_eAplZdoTransportNwkKey()** or **ZPS_eAplZdoRequestKeyReq()**.

The target node can be specified by means of its network address or IEEE/MAC address. A broadcast to multiple nodes in the network can be achieved by specifying a special network address or IEEE/MAC address - see Section 9.3.

Parent topic:ZPS_eAplZdoSwitchKeyReq

Parameters

- *u8DstAddrMode* Type of destination address:
  - ZPS_E_ADDR_MODE_SHORT - 16-bit network address.

  - ZPS_E_ADDR_MODE_IEEE - 64-bit IEEE/MAC address.
  - All other values are reserved.
- *uDstAddress* Destination address (address type as specified through *u8DstAddrMode*) - special broadcast addresses are detailed in Section 9.3.
- *u8KeySeqNum* Sequence number of new network key to adopt.

Parent topic:ZPS_eAplZdoSwitchKeyReq

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoSwitchKeyReq

Parent topic: Security functions

ZPS_eAplZdoRequestKeyReq

```
ZPS_teStatus ZPS_eAplZdoRequestKeyReq(
                uint8 u8KeyType,
                uint64 u64IeeePartnerAddr);
```

Description This function can be used to request a link key from the Trust Centre for application-level security. The possible key types that can be requested are:

- **Application link key:** This key is used to encrypt/decrypt communications with another 'partner node'. The IEEE/MAC address of this partner node must be specified as part of the function call. The Trust Centre responds by sending the application link key to both the local node and the partner node. When it arrives, the stack automatically saves this key. Also, the event ZPS_EVENT_ZDO_LINK_KEY is generated once the link key has been installed and is ready for use.
- **Trust Centre Link Key (TCLK):** This key is used to encrypt/decrypt communications between the Trust Centre and the local node. The Trust Centre responds by sending the TCLK to the requesting node.

While requesting a TCLK, the function parameter *u64IeeePartnerAddr*is ignored.

For more information on requesting link keys, refer to Section 6.8.3.2.

Parent topic:ZPS_eAplZdoRequestKeyReq

Parameters

- *u8KeyType* Type of key to request:
  - 2 - application link key
  - 4 - Trust Centre Link Key (TCLK)
  - All other values reserved
- *u64IeeePartnerAddr* IEEE/MAC address of partner node (for application link key)

Parent topic:ZPS_eAplZdoRequestKeyReq

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoRequestKeyReq

Parent topic: Security functions

ZPS_eAplZdoAddReplaceLinkKey

```
ZPS_teStatus ZPS_eAplZdoAddReplaceLinkKey(
                    uint64 u64IeeeAddr,
                    uint8 au8Key[ZPS_SEC_KEY_LENGTH],
                    ZPS_teApsLinkKeyType eKeyType);
```

Description This function can be used to introduce or replace the application link key on the local node, where this key is used to encrypt and decrypt communications with the specified 'partner node'. If an application link key already exists, then it is replaced.

The function must be called on both the local node and the partner node. Note that the Trust Centre's record of the application link key for this pair of nodes remains unchanged.

If the JCU Non-Volatile Memory Manager (NVM) module is enabled, this function also saves the application link key to Non-Volatile Memory. This allows the key to be automatically recovered during a subsequent cold start (for example, following a power failure).

The *eKeyType* parameter of this function can be used to specify 'unique' or 'global'. This does not relate to the type of key being added or replaced, which is always a unique key.

- Setting this parameter to 'unique' means that the node only ever uses the unique key.
- Setting the parameter to 'global' means that the node uses the unique key, where appropriate, and also the pre-configured global link key, where appropriate. For example, the global key would be used when another node joins the network via the local node.

Parent topic:ZPS_eAplZdoAddReplaceLinkKey

Parameters

- *u64IeeeAddr:* 64-bit IEEE/MAC address of partner node for which the specified link key is valid.
- *au8Key[]:* Array containing the link key to be added/replaced. This array has a length equal to ZPS_SEC_KEY_LENGTH.
- *eKeyType:* Type of the key to be used by the node (see above), one of the below:
  - ZPS_APS_UNIQUE_LINK_KEY, or
  - ZPS_APS_GLOBAL_LINK_KEY.

Parent topic:ZPS_eAplZdoAddReplaceLinkKey

Returns

- ZPS_E_SUCCESS (link key successfully installed)
- APS return codes, listed and described in Section 11.2.2

- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoAddReplaceLinkKey

Parent topic: Security functions

ZPS_eAplZdoAddReplaceInstallCodes

```
ZPS_teStatus ZPS_eAplZdoAddReplaceInstallCodes(
        uint64 u64IeeeAddr,
        uint8 au8InstallCode[ZPS_INSTALL_CODE_LENGTH],
        uint8 u8InstallCodeSize,
        ZPS_teApsLinkKeyType eKeyType);
```

Description This function can be used on the Trust Centre to generate a pre-configured unique link key from an install code, where this key is used to encrypt and decrypt communications between the Trust Centre and the specified node (install codes are described in the *ZigBee Devices User Guide (JNUG3131)*). If a pre-configured link key already exists for the node then it will be replaced.

The function must be called on the Trust Centre only. The other node will have the relevant pre-configured unique link key factory-installed.

If the JCU Non-Volatile Memory Manager (NVM) module is enabled, this function also saves the link key to Non-Volatile Memory. This allows the key to be automatically recovered during a subsequent cold start (for example, following a power failure).

The *eKeyType*parameter of this function can be used to specify 'unique' or 'global'. This does not relate to the type of key being added or replaced, which is always a unique key.

- Setting this parameter to 'unique' means that the Trust Centre only, ever uses the unique key with this node.
- Setting the parameter to 'global' means that the Trust Centre uses the pre-configured global link key (if available) when there is no unique link key for the node.

Parent topic:ZPS_eAplZdoAddReplaceInstallCodes

Parameters

- *u64IeeeAddr* 64-bit IEEE/MAC address of node for which the generated link key is valid.
- *au8InstallCode[]* Array containing the install code - the array length ZPS_INSTALL_CODE_LENGTH is given below in *u8InstallCodeSize*.
- *u8InstallCodeSize* Number of characters in the install code - this is the size of the array *au8InstallCode[]*.
- *eKeyType* Type of the key to be used by the node (see above), one of the below:
  - ZPS_APS_UNIQUE_LINK_KEY
  - ZPS_APS_GLOBAL_LINK_KEY

Parent topic:ZPS_eAplZdoAddReplaceInstallCodes

Returns

- ZPS_E_SUCCESS (permissions successfully obtained)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3

- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoAddReplaceInstallCodes

Parent topic: Security functions

ZPS_eAplZdoRemoveLinkKey

```
ZPS_teStatus ZPS_eAplZdoRemoveLinkKey(
                uint64 u64IeeeAddr);
```

Description This function can be used to remove the current application link key that is used to encrypt and decrypt communications between the local node and the specified 'partner node'.

The function must be called on both the local node and the partner node. Note that the Trust Centre's record of the application link key for this pair of nodes remains unchanged.

In the absence of an application link key, communications between these nodes is subsequently secured using the network key.

Parent topic:ZPS_eAplZdoRemoveLinkKey

Parameters *u64IeeeAddr:* 64-bit IEEE/MAC address of partner node for which the link key is to be removed.

Parent topic:ZPS_eAplZdoRemoveLinkKey

Returns

- ZPS_E_SUCCESS (permissions successfully removed)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoRemoveLinkKey

Parent topic: Security functions

ZPS_eAplZdoRemoveDeviceReq

```
ZPS_teStatus ZPS_eAplZdoRemoveDeviceReq(
                uint64 u64ParentAddr,
                uint64 u64ChildAddr);
```

Description This function can be used (normally by the Coordinator/Trust Centre) to request another node (such as a Router) to remove one of its children from the network (for example, if the child node does not satisfy security requirements).

The Router receiving this request ignores the request unless it has originated from the Trust Centre or is a request to remove itself. If the request was sent without APS layer encryption, the device ignores the request. If APS layer security is not in use, the alternative function **ZPS_eAplZdoLeaveNetwork()** should be used.

Parent topic:ZPS_eAplZdoRemoveDeviceReq

Parameters

- *u64ParentAddr* 64-bit IEEE/MAC address of parent to be instructed.
- *u64ChildAddr* 64-bit IEEE/MAC address of child node to be removed.

Parent topic:ZPS_eAplZdoRemoveDeviceReq

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoRemoveDeviceReq

Parent topic: Security functions

ZPS_eAplZdoSetDevicePermission

```
void ZPS_eAplZdoSetDevicePermission(
        ZPS_teDevicePermissions u8DevicePermissions);
```

Description This function can be used on any device to set the permissions for certain requests from other nodes. The possible settings are:

- Allow all requests from all other nodes (ALL_PERMITED)
- Do not allow join requests from all other nodes (JOIN_DISALLOWED)
- Do not allow data requests from all other nodes (DATA_REQUEST_DISALLOWED)

The function is particularly useful in disabling the generation of APS (end-to-end) acknowledgments, using DATA_REQUEST_DISALLOWED.

Parent topic:ZPS_eAplZdoSetDevicePermission

Parameters *u8DevicePermissions:* Bitmap of permissions to be set, constructed using the following enumerations:

- ZPS_DEVICE_PERMISSIONS_ALL_PERMITED
- ZPS_DEVICE_PERMISSIONS_JOIN_DISALLOWED
- ZPS_DEVICE_PERMISSIONS_DATA_REQUEST_DISALLOWED

Parent topic:ZPS_eAplZdoSetDevicePermission

Returns

- ZPS_E_SUCCESS (permissions successfully set)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoSetDevicePermission

Parent topic: Security functions

ZPS_bAplZdoTrustCenterSetDevicePermissions

```
ZPS_teStatus
ZPS_bAplZdoTrustCenterSetDevicePermissions(
  uint64 u64DeviceAddr,
  ZPS_teTCDevicePermissions u8DevicePermissions);
```

Description This function can be used by the Trust Centre to set the permissions for certain requests from a particular node. The possible settings are:

- Allow all requests from the specified node (ALL_PERMITED)
- Do not allow join requests from the specified node (JOIN_DISALLOWED)
- Do not allow data requests from the specified node (DATA_REQUEST_DISALLOWED)

Parent topic:ZPS_bAplZdoTrustCenterSetDevicePermissions

Parameters

- *u64DeviceAddr:* 64-bit IEEE/MAC address of node for which permissions are to be set
- *u8DevicePermissions:* Bitmap of permissions to be set, constructed using the following enumerations:
  - ZPS_TRUST_CENTER_ALL_PERMITED
  - ZPS_TRUST_CENTER_JOIN_DISALLOWED
  - ZPS_TRUST_CENTER_DATA_REQUEST_DISALLOWED

Parent topic:ZPS_bAplZdoTrustCenterSetDevicePermissions

Returns

- ZPS_E_SUCCESS (permissions successfully set)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_bAplZdoTrustCenterSetDevicePermissions

Parent topic: Security functions

ZPS_bAplZdoTrustCenterGetDevicePermissions

```
ZPS_teStatus
ZPS_bAplZdoTrustCenterGetDevicePermissions(
   uint64 u64DeviceAddr,
   ZPS_teTCDevicePermissions *pu8DevicePermissions);
```

Description This function can be used by the Trust Centre to obtain its own permissions for certain requests from a particular node. The possible settings are:

- Allow all requests from the specified node.
- Do not allow join requests from the specified node.
- Do not allow data requests from the specified node.

Parent topic:ZPS_bAplZdoTrustCenterGetDevicePermissions

Parameters

- *u64DeviceAddr:* 64-bit IEEE/MAC address of node for which permissions are to be obtained.
- *pu8DevicePermissions:* Pointer to bitmap containing permissions obtained, where:
  - 0 indicates all requests allowed.
  - 1 indicates join requests disallowed.
  - 2 indicates data requests disallowed.
  - 3 indicates data and join requests disallowed.
  - Higher bits are reserved for future use

Parent topic:ZPS_bAplZdoTrustCenterGetDevicePermissions

Returns

- ZPS_E_SUCCESS (permissions successfully obtained)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_bAplZdoTrustCenterGetDevicePermissions

Parent topic: Security functions

ZPS_bAplZdoTrustCenterRemoveDevice

```
ZPS_teStatus ZPS_bAplZdoTrustCenterRemoveDevice(
                        uint64 u64DeviceAddr);
```

Description This function can be used by the Trust Centre to delete a node in its information base.

Parent topic:ZPS_bAplZdoTrustCenterRemoveDevice

Parameters *u64DeviceAddr* : It is the 64-bit IEEE/MAC address of the node to be removed from the list.

Parent topic:ZPS_bAplZdoTrustCenterRemoveDevice

Returns

- ZPS_E_SUCCESS (node successfully removed from list)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_bAplZdoTrustCenterRemoveDevice

Parent topic: Security functions

ZPS_vTcInitFlash

```
void ZPS_vTcInitFlash(
    ZPS_tsAfFlashInfoSet *psFlashInfoSet,
    ZPS_TclkDescriptorEntry *psTclkStruct);
```

Description This function can be used on the network Coordinator/Trust Centre to enable the persistent storage of the Trust Centre Link Keys (TCLKs) for all nodes in the network.

Each of these keys is a unique application-level link key for a node. The key is used to encrypt/decrypt communications between Trust Centre and the node during the commissioning of the node into the network (and is issued by the Trust Centre to replace the pre-configured unique link key).

The function allows these link keys to be stored in devices Flash memory.

- Information about the Flash memory sector to be used to store the link keys is specified in a ZPS_tsAfFlashInfoSetstructure.
- Information about an individual link key is stored in RAM in a read-only ZPS_TclkDescriptorEntrystructure, which is for internal use by the stack. An array of these structures must be allocated in RAM, with one element for each node in the network - for example, if there are up to 250 nodes in the network, the required allocation would be:

```
ZPS_TclkDescriptorEntry sData[250];
```

The application can determine at any time whether this feature is enabled by reading the Boolean variable bSetTclkFlashFeature, which reads as TRUE if the feature is enabled and as FALSE if it is disabled.

When a new Trust Centre Link Key has been negotiated for a node, the stack on the Trust Centre notifies the application by means of a ZPS_EVENT_TC_STATUS event. The application can discover the IEEE/MAC address of the corresponding node by calling **ZPS_u64GetFlashMappedIeeeAddress()** with the value of u16ExtAddrLkupfrom the key descriptor passed in the event.

Please note that when the key table is held in RAM, **ZPS_u64NwkNibGetMappedIeeeAddr()** would be called instead.

Parent topic:ZPS_vTcInitFlash

Parameters

- *psFlashInfoSet* Pointer to a structure containing information about the Flash memory sector to be used to store the link keys for the network nodes (see Section 8.2.3.8)
- *psTclkStruct* Pointer to a structure in RAM that is used to hold information about the storage of one link key in Flash memory (see Section 8.2.3.9)

Parent topic:ZPS_vTcInitFlash

Returns None

Parent topic:ZPS_vTcInitFlash

Parent topic: Security functions

ZPS_vSetTCLockDownOverride

```
void ZPS_vSetTCLockDownOverride(
    void* pvApl,
    bool_t u8RemoteOverride,
    bool_t bDisableAuthentications);
```

Description This function can be called on the network Coordinator to disable Trust Centre functionality on the device.

The function provides two configuration options:

- Allows remote devices to over-ride the Trust Centre policy.
- Disables authentication of network joins (any transport key is also disabled).

Parent topic:ZPS_vSetTCLockDownOverride

Parameters

- *pvApl:* Handle for the relevant Application layer instance.
- *u8RemoteOverride:* Boolean specifying whether remote overrides of Trust Centre policy are to be permitted:
  - TRUE - Does not allow remote over-rides; stack does not allow the permit join remotely sent to change its local state.
  - FALSE - Allows remote over-rides; stack accepts permit join requests coming in and obeys them.
- *bDisableAuthentications:* Boolean specifying whether network join authentications are to be disabled:
  - TRUE - Disable authentications
  - FALSE - Do not disable authentications

    When this flag is set to TRUE, permit join is not accepted remotely and the TC does not transport a key to any joiner.

Parent topic:ZPS_vSetTCLockDownOverride

Returns None

Parent topic:ZPS_vSetTCLockDownOverride

Parent topic: Security functions

ZPS_psGetActiveKey

```
ZPS_tsAplApsKeyDescriptorEntry *ZPS_psGetActiveKey(
                                uint64 u64IeeeAddress,
                                uint32* pu32Index);
```

Description This function can be used on the Trust Centre to obtain the Pre-configured Unique Link Key for the node with the specified IEEE/MAC address. The function searches the local Key Descriptor Table for an entry corresponding to the specified address. If it finds a relevant entry, it returns the entry as well as the index number of the entry in the table. The required key is in the returned table entry.

Parent topic:ZPS_psGetActiveKey

Parameters

- *u64IeeeAddress:* IEEE/MAC address of the node of interest
- *pu32Index:* Pointer to a location to receive the index number of the relevant Key Descriptor Table entry

Parent topic:ZPS_psGetActiveKey

Returns Pointer to requested Key Descriptor Table entry (for structure, see Section 8.2.3.6).

Parent topic:ZPS_psGetActiveKey

Parent topic: Security functions

ZPS_vTCSetCallback

```
void ZPS_vTCSetCallback(void *pCallbackFn);
```

Description This function can be used to register a user-defined callback function on the Trust Centre, where this callback function allows the application to react to a notification from another network node - for example, to decide whether to permit a node to join that may or may not be known to the Trust Centre application.

The prototype of the user-defined callback function is:

```
bool bTransportKeyDecider (uint16 u16ShortAddress,
                           uint64 u64DeviceAddress,
                           uint64 u64ParentAddress,
                           uint8 u8Status,
                           uint16 u16Interface);
```

where:

- *u16ShortAddr* is the network address of the relevant node.
- *u64DeviceAddress* is the IEEE/MAC address of the relevant node.
- *u64ParentAddress* is the IEEE/MAC address of the parent that sent the notification.
- *u8Status* is the nature of the notification:
  - 0: Secure rejoin
  - 1: Unsecure join (association)
  - 2: Leave
  - 3: Unsecure rejoin
  - 4: Leave with a rejoin
- *u16Interface* is the MAC interface this join has happened on. If it is 2.4 G only the value is always 0. If it is a MultiMAC device 2.4 G interface, it will return value 0 and sub Gig will return value 1.

To disallow the notified action (for example, a join), the callback function should return FALSE.

If the callback function is not registered or returns TRUE, the Trust Centre will allow the notified action. In the case of a join, the Trust Centre will send the network key in a 'transport key' command to the node, either:

- encrypted with the node's pre-configured link key, if this key is known to the Trust Centre, or

- encrypted with the Trust Centre's default pre-configured link key otherwise (in this case, the joining node will only be able to decrypt the 'transport key' command and complete the join if it also has the Trust Centre's default pre-configured link key)

Registration of this callback function may be useful in controlling rejoins. A node can initially join a network using its pre-configured link key (which is also known by the Trust Centre), but this key may subsequently be replaced on the Trust Centre by an application link key (shared only by the node and the Trust Centre). If the node later leaves the network and loses its context data (including the application link key), it may attempt to rejoin the network using its pre-configured link key again. The callback function can allow the application to decide whether to permit such a rejoin. If the rejoin is to be allowed, the callback function must replace the stored application link key with the pre-configured link key on the Trust Centre before returning TRUE.

Parent topic:ZPS_vTCSetCallback

Parameters *pCallbackFn* Pointer to user-defined callback function.

Parent topic:ZPS_vTCSetCallback

Returns None

Parent topic:ZPS_vTCSetCallback

Parent topic: Security functions

Parent topic:ZDO API functions

Addressing functions The ZDO Addressing functions allow node addresses to be stored and obtained. They include the group address functions that allow a group of nodes/endpoints, with an assigned group address, to be created and modified (this group can be used as the destinations for a multicast message).

The functions are listed below.

Function page

1. ZPS_u16AplZdoGetNwkAddr
2. ZPS_u64AplZdoGetIeeeAddr
3. ZPS_eAplZdoAddAddrMapEntry
4. ZPS_u16AplZdoLookupAddr
5. ZPS_u64AplZdoLookupIeeeAddr
6. ZPS_u64NwkNibGetMappedIeeeAddr
7. ZPS_u64GetFlashMappedIeeeAddress
8. ZPS_bNwkFindAddIeeeAddr
9. ZPS_vSetOverrideLocalIeeeAddr
10. ZPS_eAplZdoGroupEndpointAdd
11. ZPS_eAplZdoGroupEndpointRemove
12. ZPS_eAplZdoGroupAllEndpointRemove

Note: Further addressing functions are provided in the ZDP API and are described in Section 9.1.1, "Address Discovery functions".

Parent topic: Addressing functions

ZPS_u16AplZdoGetNwkAddr

```
uint16 ZPS_u16AplZdoGetNwkAddr(void);
```

Description

This function obtains the 16-bit network address of the local node.

Parameters None

Parent topic:ZPS_u16AplZdoGetNwkAddr

Returns 16-bit network address obtained.

Parent topic:ZPS_u16AplZdoGetNwkAddr

Parent topic: Addressing functions

ZPS_u64AplZdoGetIeeeAddr

```
uint64 ZPS_u64AplZdoGetIeeeAddr(void);
```

Description This function obtains the 64-bit IEEE (MAC) address of the local node.

Parent topic:ZPS_u64AplZdoGetIeeeAddr

Parameters None

Parent topic:ZPS_u64AplZdoGetIeeeAddr

Returns 64-bit IEEE/MAC address obtained

Parent topic:ZPS_u64AplZdoGetIeeeAddr

Parent topic: Addressing functions

ZPS_eAplZdoAddAddrMapEntry

```
ZPS_teStatus ZPS_eAplZdoAddAddrMapEntry(
                                uint16 u16NwkAddr,
                                uint64 u64ExtAddr);
```

Description This function can be used to add the addresses of a remote node to the local Address Map table. Each entry in this table stores a remote node's 16-bit network address and an index to its 64-bit IEEE (MAC) address in the MAC Address table (see Section3.2.4). Thus, the function adds the IEEE address to the MAC Address table and then the index of this entry to the Address Map table.

Note: You should only modify the Address Map table using the supplied API functions and never write to it directly.

Parent topic:ZPS_eAplZdoAddAddrMapEntry

Parameters

- *u16NwkAddr*: 16-bit network address of node to be added
- *u64ExtAddr:* 64-bit IEEE/MAC address of node to be added

Parent topic:ZPS_eAplZdoAddAddrMapEntry

Returns

- ZPS_E_SUCCESS (addresses successfully added to tables)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4.

Parent topic:ZPS_eAplZdoAddAddrMapEntry

Parent topic: Addressing functions

ZPS_vPurgeAddressMap

```
void ZPS_vPurgeAddressMap(void);
```

Description This function removes all entries from the Address Map table on the local node.

Note: You should modify the Address Map table only using the supplied API functions and never write to it directly.

Parent topic:ZPS_vPurgeAddressMap

Parameters None.

Parent topic:ZPS_vPurgeAddressMap

Returns None.

Parent topic:ZPS_vPurgeAddressMap

Parent topic: Addressing functions

ZPS_u16AplZdoLookupAddr

```
uint16 ZPS_u16AplZdoLookupAddr(uint64 u64ExtAddr);
```

Description This function can be used to search the local Address Map table for the 16-bit network address of the node with a given 64-bit IEEE (MAC) address.

Parent topic:ZPS_u16AplZdoLookupAddr

Parameters *u64ExtAddr* 64-bit IEEE/MAC address of node to be searched for.

Parent topic:ZPS_u16AplZdoLookupAddr

Returns 16-bit network address obtained.

Parent topic:ZPS_u16AplZdoLookupAddr

Parent topic: Addressing functions

ZPS_u64AplZdoLookupIeeeAddr

```
uint64 ZPS_u64AplZdoLookupIeeeAddr(
                    uint16 u16NwkAddr);
```

Description This function can be used to search the local Address Map table for the 64-bit IEEE (MAC) address of the node with a given 16-bit network address.

Parent topic:ZPS_u64AplZdoLookupIeeeAddr

Parameters *u16NwkAddr* 16-bit network address of node to be searched for.

Parent topic:ZPS_u64AplZdoLookupIeeeAddr

Returns 64-bit IEEE/MAC address obtained.

Parent topic:ZPS_u64AplZdoLookupIeeeAddr

Parent topic: Addressing functions

ZPS_u64NwkNibGetMappedIeeeAddr

```
uint64 ZPS_u64NwkNibGetMappedIeeeAddr(
                        void *pvNwk,
                        uint16 u16Location);
```

Description This function can be used to obtain the 64-bit IEEE (MAC) address that is stored in a particular entry in the local MAC Address table. The number of the entry must be specified as well as the handle of the relevant network.

Parent topic:ZPS_u64NwkNibGetMappedIeeeAddr

Parameters

- *pvNwk:* Pointer to relevant NWK layer instance
- *u16Location:* Number of entry to access in MAC Address table

Parent topic:ZPS_u64NwkNibGetMappedIeeeAddr

Returns 64-bit IEEE/MAC address obtained.

Parent topic:ZPS_u64NwkNibGetMappedIeeeAddr

Parent topic: Addressing functions

ZPS_u64GetFlashMappedIeeeAddress

```
uint64 ZPS_u64GetFlashMappedIeeeAddress(
                    uint16 u16Location);
```

Description This function can be used on the Trust Centre to obtain the 64-bit IEEE (MAC) address of the node for which a link key has been persistently stored in the specified location in devices Flash memory. The location is specified as the number of the array element for the node - see the description of ZPS_vTcInitFlash.

Parent topic:ZPS_u64GetFlashMappedIeeeAddress

Parameters *u16Location:* Number of the array element for the node

Parent topic:ZPS_u64GetFlashMappedIeeeAddress

Returns 64-bit IEEE/MAC address obtained.

Parent topic:ZPS_u64GetFlashMappedIeeeAddress

Parent topic: Addressing functions

ZPS_bNwkFindAddIeeeAddr

```
bool_t ZPS_bNwkFindAddIeeeAddr(
            void *pvNwk,
            uint64 u64IeeeAddr,
            uint16 *pu16Location,
            bool_t bNeighborTable);
```

Description This function can be used to add the 64-bit IEEE (MAC) address of a node to the local MAC Address table. The function first searches the table to determine whether the address already exists in the table. If there is no entry for this address, a new entry for it is added to the table. The number of the entry where the address was found or added is returned in a specified location.

Note: You should modify the MAC Address table only using the supplied API functions and never write to it directly.

Parent topic:ZPS_bNwkFindAddIeeeAddr

Parameters

- *pvNwk:* Pointer to relevant NWK layer instance
- *u64IeeeAddr:* 64-bit IEEE/MAC address to be added
- *pu16Location:* Pointer to location to receive number of entry in MAC Address table where specified address was found or added
- *bNeighborTable:* Always set to FALSE

Parent topic:ZPS_bNwkFindAddIeeeAddr

Returns Boolean indicating the outcome of the operation:

- TRUE - address successfully added to the table
- FALSE - address found to already exist in the table

Parent topic:ZPS_bNwkFindAddIeeeAddr

Parent topic: Addressing functions

ZPS_vSetOverrideLocalIeeeAddr

```
void ZPS_vSetOverrideLocalIeeeAddr(
                    uint64 *pu64Address);
```

Description This function can be used to over-ride the 64-bit IEEE (MAC) address of the device where this address is stored locally in the index sector of Flash memory.

Note: If required, this function must be called before the ZigBee PRO stack is initialized.

Parent topic:ZPS_vSetOverrideLocalIeeeAddr

Parameters *pu64Address* Pointer to the 64-bit IEEE MAC address

Note: The stack stores a pointer to pu64Addressand does not take a copy of the address. The memory pointed to by pu64Address must therefore be static or constant, and must not be on the CPU stack.

Parent topic:ZPS_vSetOverrideLocalIeeeAddr

Parent topic: Addressing functions

ZPS_eAplZdoGroupEndpointAdd

```
ZPS_teStatus ZPS_eAplZdoGroupEndpointAdd(
                    uint16 u16GroupAddr,
                    uint8 u8DstEndpoint);
```

Description This function requests that the specified endpoint (on the local node) is added to the group with the specified group address. This means that this endpoint will become one of the destinations for messages sent to the given group address.

To form a group comprising endpoints from different nodes, it is necessary to call this function for each endpoint individually, on the endpoint's local node.

An endpoint can belong to more than one group.

Information on the endpoints in a group can be obtained from the Group Address table in the AIB (which can be accessed using the function **ZPS_psAplAibGetAib()**).

Note: In order to add an endpoint to a group using this function, a Group Address table must exist on the local node. This table is created using the ZPS Configuration Editor.

Parent topic:ZPS_eAplZdoGroupEndpointAdd

Parameters

- *u16GroupAddr:* 16-bit group address
- *u8DstEndpoint:* Number of destination endpoint (1-240) on local node

Parent topic:ZPS_eAplZdoGroupEndpointAdd

Returns

- ZPS_E_SUCCESS (endpoint successfully added to group)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoGroupEndpointAdd

Parent topic: Addressing functions

ZPS_eAplZdoGroupEndpointRemove

```
ZPS_teStatus ZPS_eAplZdoGroupEndpointRemove(
                        uint16 u16GroupAddr,
                        uint8 u8DstEndpoint);
```

Description This function requests that the specified endpoint (on the local node) is removed from the group with the specified group address.

If you wish to remove an endpoint from all groups to which it belongs, use the function**ZPS_eAplZdoGroupAllEndpointRemove()**.

Information on the endpoints in a group can be obtained from the Group Address table in the AIB (which can be accessed using the function **ZPS_psAplAibGetAib()**).

Parent topic:ZPS_eAplZdoGroupEndpointRemove

Parameters

- *u16GroupAddr*: 16-bit group address
- *u8DstEndpoint:* Number of destination endpoint (1-240) on local node

Parent topic:ZPS_eAplZdoGroupEndpointRemove

Returns

- ZPS_E_SUCCESS (endpoint successfully removed from group)
- APS return codes, listed and described in Section 111.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoGroupEndpointRemove

Parent topic: Addressing functions

ZPS_eAplZdoGroupAllEndpointRemove

```
ZPS_teStatus ZPS_eAplZdoGroupAllEndpointRemove(
                        uint8 u8DstEndpoint);
```

Description This function requests that the specified endpoint (on the local node) is removed from all groups to which it currently belongs.

Information on the endpoints in a group can be obtained from the Group Address table in the AIB (which can be accessed using the function **ZPS_psAplAibGetAib()**).

Parent topic:ZPS_eAplZdoGroupAllEndpointRemove

Parameters *u8DstEndpoint* Number of destination endpoint (1-240) on local node

Parent topic:ZPS_eAplZdoGroupAllEndpointRemove

Returns

- ZPS_E_SUCCESS (endpoint successfully removed from all groups)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoGroupAllEndpointRemove

Parent topic: Addressing functions

Parent topic:ZDO API functions

Routing functions The ZDO Routing functions can be used to make route discovery requests. The functions are listed below.

Function page

1. ZPS_eAplZdoRouteRequest
2. ZPS_eAplZdoManyToOneRouteRequest

Parent topic: Routing functions

ZPS_eAplZdoRouteRequest

```
ZPS_teStatus ZPS_eAplZdoRouteRequest(
                                uint16 u16DstAddr,
                                uint8 u8Radius);
```

Description This function requests the discovery of a route to the specified remote node (and that this route is added to the Routing tables in the relevant Router nodes).

Parent topic:ZPS_eAplZdoRouteRequest

Parameters

- *u16DstAddr* 16-bit network address of destination node
- *u8Radius* Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)

Parent topic:ZPS_eAplZdoRouteRequest

Returns

- ZPS_E_SUCCESS (route discovery request successfully initiated)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoRouteRequest

Parent topic: Routing functions

ZPS_eAplZdoManyToOneRouteRequest

```
ZPS_teStatus ZPS_eAplZdoManyToOneRouteRequest(
                                bool bCacheRoute,
                                uint8 u8Radius);
```

Description This function requests a 'many-to-one' route discovery and should be called on a node that will act as a 'concentrator' in the network (that is, a node with which many other nodes will need to communicate).

As a result of this function call, a route discovery message is broadcast across the network and Routing table entries (for routes back to the concentrator) are stored in the Router nodes.

The maximum number of hops to be taken by a route discovery message in this broadcast must be specified. There is also an option to store the discovered routes in a Route Record Table on the concentrator (for return communications).

Parent topic:ZPS_eAplZdoManyToOneRouteRequest

Parameters

- *bCacheRoute:* Indicates whether to store routes in Route Record Table:
  - TRUE - store routes
  - FALSE - do not store routes
- *u8Radius:* Maximum number of hops of route discovery message (zero value specifies that default maximum is to be used)

Parent topic:ZPS_eAplZdoManyToOneRouteRequest

Returns

- ZPS_E_SUCCESS (many-to-one route discovery successfully initiated)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoManyToOneRouteRequest

Parent topic: Routing functions

Parent topic:ZDO API functions

Object Handle functions The ZDO Object Handle functions can be used to obtain the handles of various objects. The functions are listed below:

Function page

1. ZPS_pvAplZdoGetAplHandle
2. ZPS_pvAplZdoGetMacHandle
3. ZPS_pvAplZdoGetNwkHandle
4. ZPS_psNwkNibGetHandle
5. ZPS_psAplAibGetAib
6. ZPS_psAplZdoGetNib

7. ZPS_u64NwkNibGetEpid

Parent topic: Object Handle functions

ZPS_pvAplZdoGetAplHandle

```
void *ZPS_pvAplZdoGetAplHandle(void);
```

Description

This function obtains a handle for the Application layer instance.

Parameters None

Parent topic:ZPS_pvAplZdoGetAplHandle

Returns Pointer to Application layer instance

Parent topic:ZPS_pvAplZdoGetAplHandle

Parent topic: Object Handle functions

ZPS_pvAplZdoGetMacHandle

```
void *ZPS_pvAplZdoGetMacHandle(void);
```

Description This function obtains a handle for the IEEE 802.15.4 MAC layer instance.

Parent topic:ZPS_pvAplZdoGetMacHandle

Parameters None

Parent topic:ZPS_pvAplZdoGetMacHandle

Returns Pointer to MAC layer instance

Parent topic:ZPS_pvAplZdoGetMacHandle

Parent topic: Object Handle functions

ZPS_pvAplZdoGetNwkHandle

```
void *ZPS_pvAplZdoGetNwkHandle(void);
```

Description This function obtains a handle for the ZigBee NWK layer instance.

Parent topic:ZPS_pvAplZdoGetNwkHandle

Parameters None

Parent topic:ZPS_pvAplZdoGetNwkHandle

Returns Pointer to NWK layer instance

Parent topic:ZPS_pvAplZdoGetNwkHandle

Parent topic: Object Handle functions

ZPS_psNwkNibGetHandle

```
ZPS_tsNwkNib *ZPS_psNwkNibGetHandle(void *pvNwk);
```

Description This function obtains a handle for the NIB (Network Information Base) corresponding to the specified NWK layer instance.

The function should be called after **ZPS_pvAplZdoGetNwkHandle()**, which is used to obtain a pointer to the NWK layer instance.

The NIB is detailed in the *ZigBee Specification (05347)* from the ZigBee Alliance. This function is not strictly a ZDO function.

Parent topic:ZPS_psNwkNibGetHandle

Parameters *pvNwk* Pointer to NWK layer instance

Parent topic:ZPS_psNwkNibGetHandle

Returns Pointer to NIB structure

Parent topic:ZPS_psNwkNibGetHandle

Example

```
void *pvNwk; = ZPS_pvAplZdoGetNwkHandle();
 ZPS_tsNwkNib *pNib = ZPS_psNwkNibGetHandle(pvNwk);
```

Parent topic:ZPS_psNwkNibGetHandle

Parent topic: Object Handle functions

ZPS_psAplAibGetAib

```
ZPS_tsAplAib *ZPS_psAplAibGetAib(void);
```

Description This function obtains a pointer to the AIB (Application Information Base) structure for the application.

Parent topic:ZPS_psAplAibGetAib

Parameters None

Parent topic:ZPS_psAplAibGetAib

Returns Pointer to AIB structure

Parent topic:ZPS_psAplAibGetAib

Parent topic: Object Handle functions

ZPS_psAplZdoGetNib

```
ZPS_tsNwkNib *ZPS_psAplZdoGetNib(void);
```

Description This function obtains a pointer to the NIB (Network Information Base) structure. The NIB is detailed in the *ZigBee Specification (05347)* from the ZigBee Alliance.

Parent topic:ZPS_psAplZdoGetNib

Parameters None

Parent topic:ZPS_psAplZdoGetNib

Returns Pointer to NIB structure

Parent topic:ZPS_psAplZdoGetNib

Parent topic: Object Handle functions

ZPS_u64NwkNibGetEpid

```
uint64 ZPS_u64NwkNibGetEpid(void *pvNwk);
```

Description This function can be used to obtain the Extended PAN ID (EPID) from a local NIB (Network Information Base).

The handle of the NWK layer instance that contains the relevant NIB must be specified. This handle can be obtained using **ZPS_pvAplZdoGetNwkHandle()**.

Parent topic:ZPS_u64NwkNibGetEpid

Parameters *pNibHandle* Pointer to NWK layer instance that contains the NIB

Parent topic:ZPS_u64NwkNibGetEpid

Returns 64-bit Extended PAN ID from NIB

Parent topic:ZPS_u64NwkNibGetEpid

Parent topic: Object Handle functions

Parent topic:ZDO API functions

Optional Cluster function The ZDO Optional Cluster function can be used to register a user-defined callback function to handle messages for a ZDO cluster that is not currently supported by the NXP ZigBee PRO stack.

The function is listed below on the function page.

Function page ZPS_eAplZdoRegisterZdoFilterCallback

Parent topic: Optional Cluster function

ZPS_eAplZdoRegisterZdoFilterCallback

```
ZPS_teStatus ZPS_eAplZdoRegisterZdoFilterCallback(
                                void *fnptr);
```

Description This function can be used to register a user-defined callback function which handles messages received for an unsupported cluster which resides on the ZDO endpoint (0), such as the cluster for an optional descriptor (for example, a user descriptor).

The prototype of the user-defined callback function is: **bool fn(uint16** clusterid **);**

where *clusterid*is the ID of the cluster that the function handles.

Normally, a message arriving for an unsupported ZDO cluster is not handled and the stack automatically returns an 'unsupported' message to the originating node. If this function is used to register a callback function for an unsupported ZDO cluster then on receiving a message for the cluster, the stack will invoke the callback function. The stack will not respond with an 'unsupported message' provided that the callback function returns TRUE, otherwise the normal stack behavior will continue.

The callback function allows the received message to be passed to the application for servicing.

Parent topic:ZPS_eAplZdoRegisterZdoFilterCallback

Parameters *fnptr:* Pointer to user-defined callback function

Parent topic:ZPS_eAplZdoRegisterZdoFilterCallback

Returns

- ZPS_E_SUCCESS (callback function successfully registered)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdoRegisterZdoFilterCallback

Parent topic: Optional Cluster function

Parent topic:ZDO API functions

Parent topic:*ZigBee Device Objects (ZDO) API*

ZDO enumerations This section details the enumerated types used by the ZDO functions. These are all defined in the header file **zps_apl_zdo.h**.

Security keys (ZPS_teZdoNwkKeyState) This structure ZPS_teZdoNwkKeyStatecontains the enumerations used to specify a type of security key:

```
typedef enum
{
ZPS_ZDO_NO_NETWORK_KEY,
ZPS_ZDO_PRECONFIGURED_LINK_KEY,
ZPS_ZDO_DISTRIBUTED_LINK_KEY,
ZPS_ZDO_PRCONFIGURED_INSTALLATION_CODE
} PACK ZPS_teZdoNwkKeyState
```

These enumerations are described in the table below:

Security Key Enumerations

Enumeration	Description
ZPS_ZDO_NO_N	No network key should be used.
ZPS_ZDO_PREC	A pre-configured link key should be used. This key can be fixed at the time of manufacture.
ZPS_ZDO_DIST	A pre-configured ZigBee Light Link (ZLL) link key should be used. This key can be fixed at the time of manufacture. A ZLL node contains both a ZPS_ZDO_PRECONFIG- URED_LINK_KEY for Home Automation (HA) compatibility and a ZPS_ZDO_ZLL_LINK_KEY for ZLL networks.
ZPS_ZDO_PRCC	A preconfigured install code is to be used. This results in a key being generated from the install code.

Parent topic:ZDO enumerations

Device types (ZPS_teZdoDeviceType) This structure ZPS_teZdoDeviceTypecontains the enumerations used to specify a ZigBee device type

```
typedef enum
{
ZPS_ZDO_DEVICE_COORD,
ZPS_ZDO_DEVICE_ROUTER,
ZPS_ZDO_DEVICE_ENDDEVICE
} PACK ZPS_teZdoDeviceType;
```

These enumerations are described in the table below.

Device Type Enumerations

Enumeration	Description
ZPS_ZDO_DEVICE_COORD	Coordinator
ZPS_ZDO_DEVICE_ROUTER	Router
ZPS_ZDO_DEVICE_ENDDEVICE	End Device

Parent topic:ZDO enumerations

Device permissions (ZPS_teDevicePermissions) This structure ZPS_teDevicePermissions contains the enumerations used on a device to specify the permissions for certain requests from other nodes:

```
typedef enum
{
ZPS_DEVICE_PERMISSIONS_ALL_PERMITED = 0,
ZPS_DEVICE_PERMISSIONS_JOIN_DISALLOWED = 1,
ZPS_DEVICE_PERMISSIONS_DATA_REQUEST_DISALLOWED = 2,
ZPS_DEVICE_PERMISSIONS_REJOIN_DISALLOWED = 4,
} PACK ZPS_teDevicePermissions;
```

These enumerations are described in the table below:

Device permissions

Enumeration	Description
ZPS_DEVICE_PERMISSIONS_ALL_PERI	Allow all requests from other nodes
ZPS_DEVICE_PERMISSIONS_JOIN_DIS/	Do not allow join requests from other nodes
ZPS_DEVICE_PERMISSIONS_DATA_RE(	Do not allow data requests from other nodes and disable end-to-end acknowledgments
ZPS_DEVICE_PERMISSIONS_REJOIN_D	Do not allow insecure rejoin.

Parent topic:ZDO enumerations

Parent topic:*ZigBee Device Objects (ZDO) API*

Application Framework (AF) API The chapter describes the resources of the Application Framework (AF) API. This API is concerned with transmitting data, controlling/monitoring local endpoints, and copying descriptors to/from the context area of the stack. The API is defined in the header file **zps_apl_af.h**.

In this chapter:

- Section 8.1 details the AF API functions.
- Section 8.2 details the AF API structures.

AF API functions The AF API functions are divided into the following categories:

- **initialization**functions, described in Section 8.1.1.
- **Data Transfer**functions, described in Section 8.1.2.
- **Endpoint**functions, described in Section 8.1.3.
- **Descriptor**functions, described in Section 8.1.4.

initialization functions The AF API contains the below initialization functions.

The functions are listed below.

1. ZPS_eAplAfInit
2. ZPS_vAplAfSetMacCapability
3. ZPS_eAplAibSetApsUseExtendedPanId
4. ZPS_vExtendedStatusSetCallback
5. ZPS_bAppAddBeaconFilter
6. ZPS_eAplFormDistributedNetworkRouter
7. ZPS_eAplInitEndDeviceDistributed
8. ZPS_vAplAfEnableMcpsFilter
9. ZPS_vNwkLinkCostCallbackRegister

Note: The function **ZPS_eAplAfInit()** is mandatory and must be the first network function called in your application.

ZPS_eAplAfInit

```
ZPS_teStatus ZPS_eAplAfInit(void);
```

Description This function initializes the Application Framework and must be the first network function called in your application code. The function first requests a reset of the Network (NWK) layer of the ZigBee PRO stack. It then initializes certain network parameters with values that have been pre-configured using the ZPS Configuration Editor (see Chapter 13, *ZPS Configuration Editor*). These parameters include the node type and the Extended PAN ID of the network.

The device is started as the pre-configured node type. If this is a Coordinator, the Extended PAN ID of the node is set to the pre-configured value. Note that if a zero value is specified, the Coordinator uses its own IEEE/MAC address for the Extended PAN ID.

Parent topic:ZPS_eAplAfInit

Parameters None.

Parent topic:ZPS_eAplAfInit

Returns

- ZPS_E_SUCCESS (AF successfully initialized).
- APS return codes, listed and described in Section 11.2.2.
- NWK return codes, listed and described in Section 11.2.3.
- MAC return codes, listed and described in Section 11.2.4.

Parent topic:ZPS_eAplAfInit

Parent topic:initialization functions

ZPS_vAplAfSetMacCapability

```
void ZPS_vAplAfSetMacCapability(uint8 u8MacCapability);
```

Description This function can be used on a Router or End Device to configure the **IEEE 802.15.4 MAC** capabilities in the Node descriptor. The MAC capabilities are specified in an 8-bit bitmap, detailed in the table below.

MAC capabilities bitmap

Bits	Description
0	Coordinator capability: 1: Node able to act as Coordinator 0: Node not able to act as Coordinator
1	Device type: 1: Full-Function Device (FFD) 0: Reduced-Function Device (RFD) An FFD can act as any node type while an RFD cannot act as the network Coordinator.
2	Power source: 1: Node is mains-powered 0: Node is not mains-powered
3	Receiver on when idle: 1: Receiver enabled during idle periods 0: Receiver disabled during idle periods to conserve power
4-5	Reserved
6	Security capability: 1: High security 0: Standard security
7	Allocate address: 1: Network address should be allocated to node 0: Network address need not be allocated to node

Parent topic:ZPS_vAplAfSetMacCapability

Parameters

- *u8MacCapability* Bitmap containing the MAC capabilities to be configured (see table above).

Parent topic:ZPS_vAplAfSetMacCapability

Returns None.

Parent topic:ZPS_vAplAfSetMacCapability

Parent topic:initialization functions

ZPS_eAplAibSetApsUseExtendedPanId

```
ZPS_teStatus ZPS_eAplAibSetApsUseExtendedPanId
                        (uint64 u64UseExtPanId);
```

Description This function can be used to create an application record of the Extended PAN ID (EPID) of the network to which the local device belongs.

- The only use of this function for a Coordinator is described in Section 6.1.1.
- The function should only be called on a Router or End Device in the manner described in Section 6.1.2.

Parent topic:ZPS_eAplAibSetApsUseExtendedPanId

Parameters *u64UseExtPanId* Extended PAN ID of network to which device belongs.

Parent topic:ZPS_eAplAibSetApsUseExtendedPanId

Returns

- ZPS_E_SUCCESS (Extended PAN ID record successfully created).
- NWK return codes, listed and described in Section 11.2.3.
- MAC return codes, listed and described in Section 11.2.4.
- APS return codes, listed and described in Section 11.2.2.

Parent topic:ZPS_eAplAibSetApsUseExtendedPanId

Parent topic:initialization functions

ZPS_vExtendedStatusSetCallback

```
void ZPS_vExtendedStatusSetCallback(
  tpfExtendedStatusCallBack pfExtendedStatusCallBack);
```

Description This function can be used to register a callback function for extended error handling (see (Section 6.7)

The prototype of the callback function is:

ZPS_teExtendedStatus vExtendedStatusCb();

The registered callback function is invoked if a subsequent API function call results in one of the following errors:

- 0xA3: ZPS_APL_APS_E_ILLEGAL_REQUEST

- 0xA6: ZPS_APL_APS_E_INVALID_PARAMETER
- 0xC2: ZPS_NWK_ENUM_INVALID_REQUEST

The callback function returns another error code (from those listed and described in Section 11.2.5), which provides a more specific reason for the error.

Parent topic:ZPS_vExtendedStatusSetCallback

Parameters *pfExtendedStatusCallBack*Pointer to extended error handling callback function to be registered.

Parent topic:ZPS_vExtendedStatusSetCallback

Returns None.

Parent topic:ZPS_vExtendedStatusSetCallback

Parent topic:ZPS_vAplAfEnableMcpsFilter

Parent topic:initialization functions

ZPS_bAppAddBeaconFilter

```
void ZPS_bAppAddBeaconFilter(tsBeaconFilterType *psAppBeaconStruct);
```

Description This function can be used to introduce a filter that will be used for filtering beacons in network searches (on a Router or End Device). Beacons can be filtered on the basis of PAN ID, Extended PAN ID, LQI value and device joining status/capacity. The filter details are provided in a tsBeaconFilterTypestructure (see Section 8.2.3.5).

If required, this function should be called immediately before **ZPS_eAplZdoDiscoverNetworks()**, **ZPS_eAplZdoRejoinNetwork()** or **ZPS_eAplZdoStartStack()**.

Note: A filter should NOT be implemented unless attempting a join, as this would prevent some stack operations from working correctly.

Once the join or discovery has completed, the filter is automatically removed and needs to be re-instated if a retry is required.

Guidelines on the implementation of beacon filters are provided in , Appendix B.4, “Beacon filtering guidelines”.

Parent topic:ZPS_bAppAddBeaconFilter

Parameters

- * *psAppBeaconStruct* Pointer to a structure containing the beacon filter details. (see Section 8.2.3.5).

Parent topic:ZPS_bAppAddBeaconFilter

Returns None.

Parent topic:ZPS_bAppAddBeaconFilter

Parent topic:initialization functions

ZPS_eAplFormDistributedNetworkRouter

```
ZPS_teStatus ZPS_eAplFormDistributedNetworkRouter(
        ZPS_tsAftsStartParamsDistributed *psStartParms,
        bool_t bSendDeviceAnnce);
```

Description This function can be used on a Router node to introduce the node into a distributed security network (see Section 6.10.2). The function must be called on the Router node that creates the distributed security network, therefore, the first node of the new network.

Subsequent Router nodes may be introduced using this function, but could be introduced using other commissioning methods, such as Touchlink.

Parent topic:ZPS_eAplFormDistributedNetworkRouter

Parameters

- *psStartParms* Pointer to structure containing the start parameter values for the Router- see Section 8.2.3.7.
- *bSendDeviceAnnce* Boolean indicating whether a device announcement message is to be sent:
  - TRUE - send device announcement
  - FALSE - do not send device announcement

Parent topic:ZPS_eAplFormDistributedNetworkRouter

Returns

- ZPS_E_SUCCESS (network successfully created).
- NWK return codes, listed and described in Section 11.2.3.
- MAC return codes, listed and described in Section 11.2.4.
- APS return codes, listed and described in Section 11.2.2.

Parent topic:ZPS_eAplFormDistributedNetworkRouter

Parent topic:initialization functions

ZPS_eAplInitEndDeviceDistributed

```
ZPS_teStatus ZPS_eAplInitEndDeviceDistributed(
            ZPS_tsAftsStartParamsDistributed *psStartParms);
```

Description This function can be used on an End Device node to introduce the node into a distributed security network (see Section 6.10.2). This network must have already been created by a Router using the **ZPS_eAplFormDistributedNetworkRouter()** function. End Device nodes may be introduced into the network in this way or using other commissioning methods, such as Touchlink.

Parent topic:ZPS_eAplInitEndDeviceDistributed

Parameters *psStartParms* Pointer to structure containing the start parameter values for the End Device - see Section 8.2.3.7, ZPS_tsAftsStartParamsDistributed.

Parent topic:ZPS_eAplInitEndDeviceDistributed

Parent topic:ZPS_vAplAfEnableMcpsFilter

Returns

- ZPS_E_SUCCESS (network successfully created).
- NWK return codes, listed and described in Section 11.2.3.
- MAC return codes, listed and described in Section 11.2.4.
- APS return codes, listed and described in Section 11.2.2.

Parent topic:ZPS_eAplInitEndDeviceDistributed

Parent topic:initialization functions

ZPS_vAplAfEnableMcpsFilter

```
void ZPS_vAplAfEnableMcpsFilter(
                bool bEnableFilter,
                uint8 u8LinkCostThreshold);
```

Description This function allows packet filtering based on 'link cost' to be enabled/disabled, as well as some basic configuration of the filtering. Packet filtering is disabled by default.

The default 'link cost threshold' is 5. This means that when packet filtering is enabled, received packets with a link cost of 5 or less are discarded by the stack and not queued for processing. The link cost threshold can be modified (from the default value of 5) using this function.

If required, this function can be called at any time after **ZPS_eAplAfInit()**.

For more information on packet filtering and link costs, refer to **Section 6.10.3**, Filtering packets on LQI Value/Link cost.

Parent topic:ZPS_vAplAfEnableMcpsFilter

Parameters *psStartParms* Pointer to structure containing the start parameter values for the End Device - see Section 8.2.3.7, ZPS_tsAftsStartParamsDistributed.

Parent topic:ZPS_eAplInitEndDeviceDistributed

Parent topic:ZPS_vAplAfEnableMcpsFilter

Returns None.

Parent topic:ZPS_vExtendedStatusSetCallback

Parent topic:ZPS_vAplAfEnableMcpsFilter

Parent topic:initialization functions

ZPS_vNwkLinkCostCallbackRegister

```
void ZPS_vNwkLinkCostCallbackRegister(void *pvFn);
```

Description This function can be used to register a user-defined callback function that defines custom mappings between LQI values and link costs that are to be used in packet filtering, based on link cost. When packet filtering is enabled, the stack uses a default set of mappings, detailed in Section 6.10.3.1. The callback function is only needed if custom mappings are to be used that will over-ride the default mappings. If required, this registration function must be called before **ZPS_eAplAfInit()**, and on both cold and warm starts.

The user-defined callback function to be registered has the following prototype:

uint8 APP_u8LinkCost(uint8 *u8Lqi*);

This callback function translates a measured LQI value (*u8Lqi*) into a link cost value. An example function is given in Section 6.10.3.3.

For more information on packet filtering and link costs, refer to Section 6.10.3.

Parent topic:ZPS_vNwkLinkCostCallbackRegister

Parameters *pvFn* Pointer to user-defined callback function to be registered.

Parent topic:ZPS_vNwkLinkCostCallbackRegister

Returns None.

Parent topic:ZPS_vNwkLinkCostCallbackRegister

Parent topic:initialization functions

Parent topic:AF API functions

Data Transfer functions The AF Data Transfer functions are used to request the transmission of data, in the form of an Application Protocol Data Unit (APDU), to one or more remote nodes.

The functions are listed below.

1. ZPS_eAplAfApsdeDataReq
2. ZPS_eAplAfUnicastDataReq
3. ZPS_eAplAfUnicastIeeeDataReq
4. ZPS_eAplAfUnicastAckDataReq
5. ZPS_eAplAfUnicastIeeeAckDataReq
6. ZPS_eAplAfGroupDataReq
7. ZPS_eAplAfBroadcastDataReq
8. ZPS_eAplAfBoundDataReq
9. ZPS_eAplAfBoundAckDataReq
10. ZPS_eAplAfInterPanDataReq
11. ZPS_u8AplGetMaxPayloadSize

Note: Functions for handling APDUs are provided in the PDUM API, described in the ***JN51xx Core Utilities User Guide (JNUG3133)***.

APDUs for Requests and Responses

A request generated by this API is sent in an APDU (Application Protocol Data Unit). A local APDU instance for the request must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()**. This function returns a handle for the APDU instance, which is subsequently used in the relevant AF API request function. Once the request has been successfully sent, the APDU instance is automatically de-allocated by the stack (there is no need for the application to de-allocate it).

Note: If the request is not successfully sent (the send function does not return ZPS_E_SUCCESS), then the APDU instance is not automatically de-allocated and the application should de-allocate it using the PDUM function PDUM_eAPduFreeAPduInstance().

When a response is subsequently received, the stack automatically allocates a local APDU instance and includes its handle in the notification event for the response. Once the response

has been dealt with, the application must de-allocate the APDU instance using the function **PDUM_eAPduFreeAPduInstance()**.

ZPS_eAplAfApsdeDataReq

```
ZPS\_teStatus ZPS\_eAplAfApsdeDataReq\(
        PDUM\_thAPduInstance hAPduInst,
        ZPS\_tsAfProfileDataReq *psProfileDataReq,
        uint8 *pu8SeqNum);
```

Description This function submits a request to send data to a remote node, with no restrictions on the type of transmission, destination address, destination application profile, destination cluster and destination endpoint number - these destination parameters do not need to be known to the stack or defined in the ZPS configuration. In this sense, this is most general of the Data Transfer functions.

The destination details and type of transmission are specified in the function call in a ZPS_tsAfProfileDataReqstructure (see Section 8.2.3.4).

The data is sent in an Application Protocol Data Unit (APDU) instance. This instance can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, this function call fails (and returns ZPS_E_ADSU_TOO_LONG). To send large APDUs, use the function **ZPS_eAplAfUnicastAckDataReq()**, which automatically implements data fragmentation (if required).

Once the sent data has reached the first hop node in the route to its destination, a ZPS_EVENT_APS_DATA_CONFIRM event is generated on the local node.

Parent topic:ZPS_eAplAfApsdeDataReq

Parameters

- ***hAPduInst**:* Handle of APDU instance to be sent.
- **psProfileDataReq:* Pointer to structure containing the details for the transmission (see Section 8.2.3.4).
- ***pu8SeqNum**:* Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL.

Parent topic:ZPS_eAplAfApsdeDataReq

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfApsdeDataReq

Parent topic: Data Transfer functions

ZPS_eAplAfUnicastDataReq

```
ZPS\_teStatus ZPS\_eAplAfUnicastDataReq\(
        PDUM\_thAPduInstance hAPduInst,
        uint16 u16ClusterId,
        uint8 u8SrcEndpoint,
        uint8 u8DstEndpoint,
        uint16 u16DestAddr,
        ZPS\_teAplAfSecurityMode eSecurityMode,
        uint8 u8Radius,
        uint8 *pu8SeqNum);
```

Description This function submits a request to send data to a remote node (unicast), using the remote node's network address. You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint), as well as the network address of the remote node and the destination endpoint on the node.

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, this function call will fail (and return ZPS_E_ADSU_TOO_LONG). To send large APDUs, use the function **ZPS_eAplAfUnicastAckDataReq()**, which automatically implements data fragmentation (if required).

Once the sent data has reached the first hop node in the route to its destination, a ZPS_EVENT_APS_DATA_CONFIRM event will be generated on the local node.

If data is sent using this function to a destination for which a route has not already been established, the data will not be sent and a route discovery will be performed instead. In this case, the function will return ZPS_NWK_ENUM_ROUTE_ERROR and must later be re-called to send the data (see Note under Section 6.5.1.1, "Unicast").

Security (encryption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parent topic:ZPS_eAplAfUnicastDataReq

Parameters

- ***hAPduInst:*** Handle of APDU instance to be sent
- ***u16ClusterId:*** Identifier of relevant output cluster on source endpoint
- ***u8SrcEndpoint:*** Source endpoint number (1-240) on local node
- ***u8DstEndpoint***: Destination endpoint number (1-240) on remote node
- ***u16DstAddr:*** Network address of destination node
- ***eSecurityMode:*** Security mode for data transfer:
  - ZPS_E_APL_AF_UNSECURE (no security enabled)
  - ZPS_E_APL_AF_SECURE (Application-level security using link key and network key)
  - ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key)
  - ZPS_E_APL_AF_SECURE | ZPS_E_APL_AF_EXT_NONCE (Application-level security using link key and network key with the extended NONCE included in the frame)
  - ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wildcard profile (0xFFFF) instead of the profile in the associated Simple descriptor)

- ***u8Radius**:* Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)
- ****pu8SeqNum**:* Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL

Parent topic:ZPS_eAplAfUnicastDataReq

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfUnicastDataReq

Parent topic: Data Transfer functions

ZPS_eAplAfUnicastIeeeDataReq

```
ZPS\_teStatus ZPS\_eAplAfUnicastIeeeDataReq(
        PDUM\_thAPduInstance hAPduInst,
        uint16 u16ClusterId,
        uint8 u8SrcEndpoint,
        uint8 u8DstEndpoint,
        uint64 u64DestAddr,
        ZPS\_teAplAfSecurityMode eSecurityMode,
        uint8 u8Radius,
        uint8 *pu8SeqNum);
```

Description This function submits a request to send data to a remote node (unicast), using the remote node's IEEE (MAC) address. You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint), as well as the IEEE address of the remote node and the destination endpoint on the node.

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, this function call will fail (and return ZPS_E_ADSU_TOO_LONG). To send large APDUs, use the function **ZPS_eAplAfUnicastIeeeAckDataReq()**, which automatically implements data fragmentation (if required).

Once the sent data has reached the first hop node in the route to its destination, a ZPS_EVENT_APS_DATA_CONFIRM event is generated on the local node.

If users try to send data using this function to a destination for which a route has not already been established, the data is not sent. Instead, a route discovery is performed. In this case, the function returns ZPS_NWK_ENUM_ROUTE_ERROR and must later be re-called to send the data (see Note under Section 6.5.1.1, "Unicast").

Security (encryption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parent topic:ZPS_eAplAfUnicastIeeeDataReq

Parameters

- ***hAPduInst:*** Handle of APDU instance to be sent
- ***u16ClusterId:*** Identifier of relevant output cluster on source endpoint
- ***u8SrcEndpoint:*** Source endpoint number (1-240) on local node
- ***u8DstEndpoint*:** Destination endpoint number (1-240) on remote node
- ***u64DestAddr:*** IEEE (MAC) address of destination node
- ***eSecurityMode:*** Security mode for data transfer:
  - ZPS_E_APL_AF_UNSECURE (no security enabled)
  - ZPS_E_APL_AF_SECURE (Application-level security using link key and network key)
  - ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key)
  - ZPS_E_APL_AF_SECURE | ZPS_E_APL_AF_EXT_NONCE (Application-level security using link key and network key with the extended NONCE included in the frame)
  - ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wildcard profile (0xFFFF) instead of the profile in the associated Simple descriptor)
- ***u8Radius:*** Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)
- ****pu8SeqNum:*** Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL

Parent topic:ZPS_eAplAfUnicastIeeeDataReq

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfUnicastIeeeDataReq

Parent topic: Data Transfer functions

ZPS_eAplAfUnicastAckDataReq

```
ZPS\_teStatus ZPS\_eAplAfUnicastAckDataReq(
            PDUM\_thAPduInstance hAPduInst,
            uint16 u16ClusterId,
            uint8 u8SrcEndpoint,
            uint8 u8DstEndpoint,
            uint16 u16DestAddr,
            ZPS\_teAplAfSecurityMode eSecurityMode,
            uint8 u8Radius,
            uint8 *pu8SeqNum);
```

Description This function submits a request to send data to a remote node (unicast), using the remote node's network address, and requires an acknowledgment to be returned by the remote node once the data reaches its destination. You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint), as well as the network address of the remote node and the destination endpoint on the node.

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, the APDU is broken up into fragments (NPDUs) for transmission. For this to happen, users should enable fragmentation by setting the ZigBee network parameter *Maximum Number of Transmitted Simultaneous Fragmented Messages* to a non-zero value.

If data is sent using this function to a destination for which a route has not already been established, the data fails to send and a route discovery is performed instead. In this case, the function returns ZPS_NWK_ENUM_ROUTE_ERROR and must later be re-called to send the data (see Note under Section 6.5.1.1, "Unicast").

Once the sent data has reached the first hop node in the route to its destination, a ZPS_EVENT_APS_DATA_CONFIRM event is generated on the local node. Then, once an acknowledgment has been received from the destination node, a ZPS_EVENT_APS_DATA_ACK is generated on the sending node.

Security (encyption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parent topic:ZPS_eAplAfUnicastAckDataReq

Parameters

- ***hAPduInst*** Handle of APDU instance to be sent
- ***u16ClusterId*** Identifier of relevant output cluster on source endpoint
- ***u8SrcEndpoint*** Source endpoint number (1-240) on local node
- ***u8DstEndpoint*** Destination endpoint number (1-240) on remote node
- ***u16DstAddr*** Network address of destination node
- ***eSecurityMode*** Security mode for data transfer:
  - ZPS_E_APL_AF_UNSECURE (no security enabled)
  - ZPS_E_APL_AF_SECURE (Application-level security using link key and network key)
  - ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key)
  - ZPS_E_APL_AF_SECURE | ZPS_E_APL_AF_EXT_NONCE (Application-level security using link key and network key with the extended NONCE included in the frame)
  - ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wildcard profile (0xFFFF) instead of the profile in the associated Simple descriptor)
- ***u8Radius*** Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used).
- ****pu8SeqNum*** Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL.

Parent topic:ZPS_eAplAfUnicastAckDataReq

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfUnicastAckDataReq

Parent topic: Data Transfer functions

ZPS_eAplAfUnicastIeeeAckDataReq

```
ZPS\_teStatus ZPS\_eAplAfUnicastIeeeAckDataReq\(
        PDUM\_thAPduInstance hAPduInst,
        uint16 u16ClusterId,
        uint8 u8SrcEndpoint,
        uint8 u8DstEndpoint,
        uint64 u64DestAddr,
        ZPS\_teAplAfSecurityMode eSecurityMode,
        uint8 u8Radius,
        uint8 *pu8SeqNum);
```

Description This function submits a request to send data to a remote node (unicast), using the remote node's IEEE (MAC) address. The function also requires an acknowledgment to be returned by the remote node once the data reaches its destination. You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint), as well as the IEEE address of the remote node and the destination endpoint on the node.

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, the APDU can be broken up into fragments (NPDUs) for transmission. To enable this fragmentation, users should set the ZigBee network parameter *Maximum Number of Transmitted Simultaneous Fragmented Messages*to a non-zero value.

If data is sent using this function to a destination for which a route has not already been established, the data is not sent and a route discovery is performed instead. In this case, the function returns ZPS_NWK_ENUM_ROUTE_ERROR and must later be re-called to send the data (see Note under Section 6.5.1.1, "Unicast").

Once the sent data has reached the first hop node in the route to its destination, a ZPS_EVENT_APS_DATA_CONFIRM event will be generated on the local node. Then, once an acknowledgment has been received from the destination node, a ZPS_EVENT_APS_DATA_ACK is generated on the sending node.

Security (encyption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parent topic:ZPS_eAplAfUnicastIeeeAckDataReq

Parameters

- ***hAPduInst:*** Handle of APDU instance to be sent
- ***u16ClusterId:*** Identifier of relevant output cluster on source endpoint
- ***u8SrcEndpoint:*** Source endpoint number (1-240) on local node
- ***u8DstEndpoint:*** Destination endpoint number (1-240) on remote node
- ***u64DestAddr:*** IEEE (MAC) address of destination node
- ***eSecurityMode:*** Security mode for data transfer:
  - ZPS_E_APL_AF_UNSECURE (no security enabled)
  - ZPS_E_APL_AF_SECURE (Application-level security using link key and network key)

  - ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key)
  - ZPS_E_APL_AF_SECURE | ZPS_E_APL_AF_EXT_NONCE (Application-level security using link key and network key with the extended NONCE included in the frame)
  - ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wildcard profile (0xFFFF) instead of the profile in the associated Simple descriptor)
- ***u8Radius:*** Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used).
- ****pu8SeqNum:*** Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL.

Parent topic:ZPS_eAplAfUnicastIeeeAckDataReq

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfUnicastIeeeAckDataReq

Parent topic: Data Transfer functions

ZPS_eAplAfGroupDataReq

```
ZPS\_teStatus ZPS\_eAplAfGroupDataReq\(
        PDUM\_thAPduInstance hAPduInst,
        uint16 u16ClusterId,
        uint8 u8SrcEndpoint,
        uint16 u16DstGroupAddr,
        ZPS\_teAplAfSecurityMode eSecurityMode,
        uint8 u8Radius,
        uint8 *pu8SeqNum);
```

Description This function submits a request to send data to a group of endpoints located on one or more nodes (group multicast). Users must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint) as well as the 'group address' of the group of destination endpoints. A group is set up using the function **ZPS_eAplZdoGroupEndpointAdd()**. The data is actually broadcast to all network nodes and each recipient node assesses whether it has endpoints in the specified group.

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, this function call fails (and returns ZPS_E_ADSU_TOO_LONG). Once the data is transmitted, a ZPS_EVENT_APS_DATA_CONFIRM event is generated on the local node.

Security (encyption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parent topic:ZPS_eAplAfGroupDataReq

Parameters

- ***hAPduInst:*** Handle of APDU instance to be sent
- ***u16ClusterId:*** Identifier of relevant output cluster on source endpoint
- ***u8SrcEndpoint:*** Source endpoint number (1-240) on local node
- ***u16DstGroupAddr:*** Group address of destination endpoints
- ***eSecurityMode:*** Security mode for data transfer, one of:
  - ZPS_E_APL_AF_UNSECURE (no security enabled)
  - ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key)
  - ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wildcard profile (0xFFFF) instead of the profile in the associated Simple descriptor)
- ***u8Radius:*** Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)
- ****pu8SeqNum:*** Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL.

Parent topic:ZPS_eAplAfGroupDataReq

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfGroupDataReq

Parent topic: Data Transfer functions

ZPS_eAplAfBroadcastDataReq

```
ZPS\_teStatus ZPS\_eAplAfBroadcastDataReq\(
    PDUM\_thAPduInstance hAPduInst,
    uint16 u16ClusterId,
    uint8 u8SrcEndpoint,
    uint8 u8DstEndpoint,
    ZPS\_teAplAfBroadcastMode eBroadcastMode,
    ZPS\_teAplAfSecurityMode eSecurityMode,
    uint8 u8Radius,
    uint8 *pu8SeqNum);
```

Description This function submits a request to send data to all network nodes that conform to the specified broadcast mode. You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint), as well as the destination endpoint(s) on the remote nodes.

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, this function call fails (and return ZPS_E_ADSU_TOO_LONG).

Following this function call, the APDU may be broadcast up to four times by the source node (in addition, the APDU may be subsequently re-broadcast up to four times by each intermediate routing node). If the transmission is successful, the event ZPS_EVENT_APS_DATA_CONFIRM is generated on the local node.

Security (encyption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parent topic:ZPS_eAplAfBroadcastDataReq

Parameters

- ***hAPduInst:*** Handle of APDU instance to be sent
- ***u16ClusterId:*** Identifier of relevant output cluster on source endpoint
- ***u8SrcEndpoint:*** Source endpoint number (1-240) on local node
- ***u8DstEndpoint:*** Destination endpoint number (1-240) on remote node, or 255 for all endpoints on node
- ***eBroadcastMode:*** Type of broadcast, one of:
  - ZPS_E_BROADCAST_ALL (all nodes)
  - ZPS_E_BROADCAST_ALL RX_ON (all nodes with radio receiver permanently enabled)
  - ZPS_E_BROADCAST_ZC_ZR (all Routers and Coordinator)
- ***eSecurityMode:*** Security mode for data transfer:
  - ZPS_E_APL_AF_UNSECURE (no security enabled)
  - ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key)
  - ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wildcard profile (0xFFFF) instead of the profile in the associated Simple descriptor)
- ***u8Radius:*** Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)
- ****pu8SeqNum:*** Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL

Parent topic:ZPS_eAplAfBroadcastDataReq

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfBroadcastDataReq

Parent topic: Data Transfer functions

ZPS_eAplAfBoundDataReq

```
ZPS\_teStatus ZPS\_eAplAfBoundDataReq\(
        PDUM\_thAPduInstance hAPduInst,
        uint16 u16ClusterId,
        uint8 u8SrcEndpoint,
```

(continues on next page)

(continued from previous page)

```
ZPS\__teAplAfSecurityMode eSecurityMode,
uint8 u8Radius,
uint8 *pu8SeqNum);
```

Description This function submits a request to send data to all nodes/endpoints to which the source node/endpoint has been previously bound (using the binding functions, described in Section 9.1.3). You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint).

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, this function call fails (and return ZPS_E_ADSU_TOO_LONG).

Once the sent data has reached the first hop node in the route to its destination(s), a ZPS_EVENT_BIND_REQUEST_SERVER event is generated on the local node. This event reports the status of the bound transmission, including the number of bound endpoints for which the transmission has failed.

Security (encyption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parent topic:ZPS_eAplAfBoundDataReq

Parameters

- ***hAPduInst:*** Handle of APDU instance to be sent
- ***u16ClusterId:*** Identifier of relevant output cluster on source endpoint
- ***u8SrcEndpoint:*** Source endpoint number (1-240) on local node
- ***eSecurityMode:*** Security mode for data transfer:
  - ZPS_E_APL_AF_UNSECURE (no security enabled)
  - ZPS_E_APL_AF_SECURE (Application-level security using link key and network key)
  - ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key)
  - ZPS_E_APL_AF_SECURE | ZPS_E_APL_AF_EXT_NONCE (Application-level security using link key and network key with the extended NONCE included in the frame)
  - ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wildcard profile (0xFFFF) instead of the profile in the associated Simple descriptor)
- ***u8Radius:*** Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)
- ****pu8SeqNum:*** Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL.

Parent topic:ZPS_eAplAfBoundDataReq

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3

- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfBoundDataReq

Parent topic: Data Transfer functions

ZPS_eAplAfBoundAckDataReq

```
ZPS\__teStatus ZPS\__eAplAfBoundAckDataReq(
            PDUM\__thAPduInstance hAPduInst,
            uint16 u16ClusterId,
            uint8 u8SrcEndpoint,
            ZPS\__teAplAfSecurityMode eSecurityMode,
            uint8 u8Radius,
            uint8 *pu8SeqNum);
```

Description This function submits a request to send data to all nodes/endpoints to which the source node/endpoint has been previously bound (using the binding functions, described in Section 9.1.3) and requires an acknowledgment to be returned by the remote node(s) once the data reaches its destination(s). You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint).

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, the APDU can be broken up into fragments (NPDUs) for transmission. To enable this fragmentation, set the ZigBee network parameter *Maximum Number of Transmitted Simultaneous Fragmented Messages* to a non-zero value.

Once the sent data reaches its final destination node(s), a ZPS_EVENT_BIND_REQUEST_SERVER event is generated on the local node. This event reports the status of the bound transmission, including the number of bound endpoints for which the transmission has failed.

Security (encyption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parent topic:ZPS_eAplAfBoundAckDataReq

Parameters

- ***hAPduInst:*** Handle of APDU instance to be sent
- ***u16ClusterId:*** Identifier of relevant output cluster on source endpoint
- ***u8SrcEndpoint:*** Source endpoint number (1-240) on local node
- ***eSecurityMode:*** Security mode for data transfer:
  - ZPS_E_APL_AF_UNSECURE (no security enabled)
  - ZPS_E_APL_AF_SECURE (Application-level security using link key and network key)
  - ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key)
  - ZPS_E_APL_AF_SECURE | ZPS_E_APL_AF_EXT_NONCE (Application-level security using link key and network key with the extended NONCE included in the frame)
  - ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wildcard profile (0xFFFF) instead of the profile in the associated Simple descriptor)
- ***u8Radius:*** Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)

- ***pu8SeqNum:*** Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL.

Parent topic:ZPS_eAplAfBoundAckDataReq

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfBoundAckDataReq

Parent topic: Data Transfer functions

ZPS_eAplAfInterPanDataReq

```
ZPS\__teStatus ZPS\__eAplAfInterPanDataReq\(
                PDUM\__thAPduInstance hAPduInst,
                uint16 u16ClusterId,
                uint16 u16ProfileId,
                ZPS\__tsInterPanAddress *psDstAddr,
                uint8 u8Handle);
```

Description This function submits a request to send data to one or more nodes in another ZigBee PRO network - that is, to implement an inter-PAN transmission. The destination for the data is specified in a structure (detailed in Section 8.2.3.3) which contains:

- PAN ID of destination network (a broadcast to all reachable ZigBee PRO networks can also be configured)
- Address of destination node (this can be an IEEE/MAC or network address for a single node, a group address for multiple nodes or a broadcast address for all nodes).

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the local network, this function call fails (and returns ZPS_E_ADSU_TOO_LONG).

Once the sent data reaches the first hop node in the route to its destination, a ZPS_EVENT_APS_INTERPAN_DATA_CONFIRM event is generated on the local node. In case of a broadcast or group multicast, this event is simply generated once the data has been sent from the local node.

Security (encyption/decryption) cannot be applied to inter-PAN transmissions.

Parent topic:ZPS_eAplAfInterPanDataReq

Parameters

- ***hAPduInst:*** Handle of APDU instance to be sent
- ***u16ClusterId:*** Identifier of cluster for which data is intended at destination (must be a cluster of the application profile specified below)
- ***u16ProfileId:*** Identifier of application profile for which data is intended at destination

- ***psDstAddr:*** Pointer to structure containing destination PAN ID and address (see Section 8.2.3.3)
- ***u8Handle:*** Handle for internal use (set to any value)

Parent topic:ZPS_eAplAfInterPanDataReq

Returns

- ZPS_E_SUCCESS.
- ZPS_APL_APS_E_ILLEGAL_REQUEST.
- MAC return codes, listed and described in Section 11.2.4.

Parent topic:ZPS_eAplAfInterPanDataReq

Parent topic: Data Transfer functions

ZPS_u8AplGetMaxPayloadSize

```
uint8 ZPS_u8AplGetMaxPayloadSize(void *pvApl,
                                 uint16 u16Addr);
```

Description This function obtains the effective payload size, in bytes, within an IEEE802.15.4 data frame to be sent to the node with the specified network address. The handle of the relevant Application layer instance must also be specified, which can be obtained using **ZPS_pvAplZdoGetAplHandle()**.

An IEEE802.15.4 data frame contains 127 bytes, but the effective payload is reduced by the various IEEE802.15.4 and ZigBee headers. The function returns the size of the payload available for data but does not take into account bytes needed for ZCL cluster headers (so may not reflect the exact amount of space available for data).

Parent topic:ZPS_u8AplGetMaxPayloadSize

Parameters

- ***pvApl***: Handle of handle for the Application layer instance
- ***u16Addr:*** 16-bit network address of node to which data is to be sent

Parent topic:ZPS_u8AplGetMaxPayloadSize

Returns Number of data frame payload bytes available for data (ignoring ZCL headers).

Parent topic:ZPS_u8AplGetMaxPayloadSize

Parent topic: Data Transfer functions

Parent topic:AF API functions

Endpoint functions The AF Endpoint functions are used to control and monitor the states of endpoints on the local node.

The functions are listed below.

1. ZPS_vAplAfSetEndpointState
2. ZPS_eAplAfGetEndpointState
3. ZPS_eAplAfSetEndpointDiscovery

4. ZPS_eAplAfGetEndpointDiscovery

ZPS_vAplAfSetEndpointState

```
ZPS\_teStatus ZPS\_eAplAfSetEndpointState\(
                                uint8 u8Endpoint,
                                bool bEnabled);
```

Description This function puts the specified endpoint on the local node into the specified state (enabled or disabled).

Parent topic:ZPS_vAplAfSetEndpointState

Parameters

- ***u8Endpoint:*** Endpoint number (on local node)
- ***bEnabled:*** State in which to put endpoint, one of:
  - TRUE: enable endpoint
  - FALSE: disable endpoint

Parent topic:ZPS_vAplAfSetEndpointState

Returns

- ZPS_E_SUCCESS (endpoint state successfully set)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_vAplAfSetEndpointState

Parent topic: Endpoint functions

ZPS_eAplAfGetEndpointState

```
ZPS\_teStatus ZPS\_eAplAfGetEndpointState\(
                                uint8 u8Endpoint,
                                bool *pbEnabled);
```

Description This function obtains the current state (enabled or disabled) of the specified endpoint on the local node.

Parent topic:ZPS_eAplAfGetEndpointState

Parameters

- ***u8Endpoint***: Endpoint number (on local node)
- ****pbEnabled***: Pointer to location to receive endpoint state. The returned state is one of:
  - TRUE: endpoint enabled
  - FALSE: endpoint disabled

Parent topic:ZPS_eAplAfGetEndpointState

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfGetEndpointState

Parent topic: Endpoint functions

ZPS_eAplAfSetEndpointDiscovery

```
ZPS\_teStatus ZPS\_eAplAfSetEndpointDiscovery\(
                                uint8 u8Endpoint,
                                uint16 u16ClusterId,
                                bool bOutput,
                                bool bDiscoverable);
```

Description This function sets the discoverable state of the specified cluster of the specified endpoint on the local node - that is, whether the cluster/endpoint will be included in 'device discoveries' initiated on the network.

If the cluster/endpoint is discoverable, it appears in the Simple descriptor of the local node and is also included in match results requested using the function **ZPS_eAplZdpMatchDescRequest()**.

The initial discoverable state of the cluster/endpoint is pre-set using the ZPS Configuration Editor (see *Chapter 13*).

Parent topic:ZPS_eAplAfSetEndpointDiscovery

Parameters

- ***u8Endpoint:*** Endpoint number (on local node)
- ***u16ClusterId:*** Cluster ID
- ***bOutput:*** Type of cluster (output or input), one of:
  - TRUE: Output cluster
  - FALSE: Input cluster
- ***bDiscoverable:*** Discoverable state to set, one of:
  - TRUE: Discoverable
  - FALSE: Not discoverable

Parent topic:ZPS_eAplAfSetEndpointDiscovery

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfSetEndpointDiscovery

Parent topic: Endpoint functions

ZPS_eAplAfGetEndpointDiscovery

```
ZPS\_teStatus ZPS\_eAplAfGetEndpointDiscovery\(
                                uint8 u8Endpoint,
                                uint16 u16ClusterId,
                                bool bOutput,
                                bool\_t *pbDiscoverable);
```

Description This function obtains the discoverable state of the specified cluster of the specified endpoint on the local node - that is, whether the cluster/endpoint will be included in 'device discoveries' initiated on the network.

If the cluster/endpoint is discoverable, it appears in the Simple descriptor of the local node and is also included in match results requested using the function **ZPS_eAplZdpMatchDescRequest()**.

The initial discoverable state of the cluster/endpoint is pre-set using the ZPS Configuration Editor (see *Chapter 13*). The state can subsequently be changed at runtime using the function **ZPS_eAplAfSetEndpointDiscovery()**.

Parent topic:ZPS_eAplAfGetEndpointDiscovery

Parameters

- ***u8Endpoint**:* Endpoint number (on local node)
- ***u16ClusterId***: Cluster ID
- ***bOutput***: Type of cluster (output or input), one of:
  - TRUE: Output cluster
  - FALSE: Input cluster
- **pbDiscoverable:* Pointer to location to receive discoverable state, which is one of the below:
  - TRUE: Discoverable
  - FALSE: Not discoverable

Parent topic:ZPS_eAplAfGetEndpointDiscovery

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfGetEndpointDiscovery

Parent topic: Endpoint functions

Parent topic:AF API functions

Descriptor functions The AF Descriptor functions allow ZigBee descriptors for the local node to be copied to and from the context area of the ZigBee PRO stack. The functions are listed below.

1. ZPS_eAplAfGetNodeDescriptor
2. ZPS_eAplAfGetNodePowerDescriptor
3. ZPS_eAplAfGetSimpleDescriptor

ZPS_eAplAfGetNodeDescriptor

```
ZPS\_teStatus ZPS\_eAplAfGetNodeDescriptor\(
        ZPS\_tsAplAfNodeDescriptor *psDesc);
```

Description This function copies the Node descriptor (for the local node) from the context area of the stack to the specified structure (the descriptor is returned through the function's parameter).

Parent topic:ZPS_eAplAfGetNodeDescriptor

Parameters * *psDesc*: Pointer to structure (see Section 8.2.1.1) to receive Node descriptor.

Parent topic:ZPS_eAplAfGetNodeDescriptor

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfGetNodeDescriptor

Parent topic: Descriptor functions

ZPS_eAplAfGetNodePowerDescriptor

```
ZPS\_teStatus ZPS\_eAplAfGetNodePowerDescriptor\(
        ZPS\_tsAplAfNodePowerDescriptor *psDesc);
```

Description This function copies the Node Power descriptor (for the local node) from the context area of the stack to the specified structure (the descriptor is returned through the function's parameter).

Parent topic:ZPS_eAplAfGetNodePowerDescriptor

Parameters * *psDesc* Pointer to structure (see Section 8.2.1.2) to receive Node Power descriptor

Parent topic:ZPS_eAplAfGetNodePowerDescriptor

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfGetNodePowerDescriptor

Parent topic: Descriptor functions

ZPS_eAplAfGetSimpleDescriptor

```
ZPS\__teStatus ZPS\__eAplAfGetSimpleDescriptor\(
        uint8 u8Endpoint,
        ZPS\__tsAplAfSimpleDescriptor *psDesc);
```

Description This function copies the Simple descriptor for the specified endpoint (on the local node) from the context area of the stack to the specified structure (the descriptor is returned through the function's parameter).

Parent topic:ZPS_eAplAfGetSimpleDescriptor

Parameters * *psDesc* Pointer to structure (see Section 8.2.1.3) to receive Simple descriptor

Parent topic:ZPS_eAplAfGetSimpleDescriptor

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfGetSimpleDescriptor

Parent topic: Descriptor functions

Parent topic:AF API functions

Other functions This section described other functions in the AF API. These functions are listed below:

- ZPS_vSaveAllZpsRecords
- ZPS_bAplAfSetEndDeviceTimeout
- ZPS_eAplAfSendKeepAlive

ZPS_vSaveAllZpsRecords

```
void ZPS\__vSaveAllZpsRecords\(void\);
```

Description This function saves to Non-Volatile Memory (NVM) all the NVM records related to the ZigBee PRO stack. This function must be used in conjunction with the Non-Volatile Memory Manager (NVM), which is described in the *JN51xx Core Utilities User Guide (JNUG3133)*.

Parent topic:ZPS_vSaveAllZpsRecords

Parameters None

Parent topic:ZPS_vSaveAllZpsRecords

Returns None

Parent topic:ZPS_vSaveAllZpsRecords

Parent topic: Other functions

ZPS_bAplAfSetEndDeviceTimeout

```
bool ZPS\__bAplAfSetEndDeviceTimeout
            \(teZedTimeout eZedTimeout);
```

Description This function can be used on an End Device to configure a timeout period for the End Device Aging mechanism, which is described in Section 6.10.1.

The End Device communicates this timeout period to its parent on joining the network. The parent applies this timeout to the 'keep-alive' packets sent from the End Device child using the function **ZPS_eAplAfSendKeepAlive()**. If the parent does not receive a keep-alive packet from the End Device before the timeout expires, then the parent assumes the End Device is no longer active and discards it.

Parent topic:ZPS_bAplAfSetEndDeviceTimeout

Parameters *eZedTimeout* Enumeration indicating timeout period to be set - one of the below:

- ZED_TIMEOUT_10_SEC (10 seconds)
- ZED_TIMEOUT_2_MIN (2 minutes)
- ZED_TIMEOUT_4_MIN (4 minutes)
- ZED_TIMEOUT_8_MIN (8 minutes)
- ZED_TIMEOUT_16_MIN (16 minutes)
- ZED_TIMEOUT_32_MIN (32 minutes)
- ZED_TIMEOUT_64_MIN (64 minutes)
- ZED_TIMEOUT_128_MIN (128 minutes)
- ZED_TIMEOUT_256_MIN (256 minutes)
- ZED_TIMEOUT_512_MIN (512 minutes)
- ZED_TIMEOUT_1024_MIN (1024 minutes)
- ZED_TIMEOUT_2048_MIN (2048 minutes)
- ZED_TIMEOUT_4096_MIN (4096 minutes)
- ZED_TIMEOUT_8192_MIN (8192 minutes)
- ZED_TIMEOUT_16384_MIN (16384 minutes)

Parent topic:ZPS_bAplAfSetEndDeviceTimeout

Returns

- TRUE - timeout successfully set
- FALSE - timeout not set

Parent topic:ZPS_bAplAfSetEndDeviceTimeout

Parent topic: Other functions

ZPS_eAplAfSendKeepAlive

```
ZPS\__teStatus ZPS\__eAplAfSendKeepAlive\(void\);
```

Description This function can be used on an End Device to send a 'keep-alive' packet to its parent as part of the End Device Aging mechanism, which is described in Section 6.10.1. This packet informs the parent that the End Device is still active, so that the parent does not discard the child.

The parent must receive at least one keep-alive packet from the End Device within the timeout period defined using the function **ZPS_bAplAfSetEndDeviceTimeout()**. Otherwise, the parent assumes that the child is no longer active and discard the child. It is recommended that at least three keep-alive packets are sent within the timeout period to ensure that the End Device child is not accidentally discarded due to missed keep-alive packets.

A keep-alive packet can take the form of a MAC Data Poll or an End Device Timeout Request, as required by the parent - the keep-alive packet type is configured in the NIB on the parent but, by default, both packets types are configured to be acceptable in the NXP software. This function automatically sends the appropriate keep-alive packet type but when both packet types are acceptable, a Data Poll is sent. Both packet types have the effect of re-starting the timeout for the End Device on the parent. When a Data Poll packet is used, the parent may also return pending data to the End Device, indicated by a ZPS_EVENT_AF_DATA_INDICATION event on the End Device.

Parent topic:ZPS_eAplAfSendKeepAlive

Parameters None

Parent topic:ZPS_eAplAfSendKeepAlive

Returns

- ZPS_E_SUCCESS
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplAfSendKeepAlive

Parent topic: Other functions

Parent topic:AF API functions

Parent topic:*Application Framework (AF) API*

AF structures This section describes the structures of the Application Framework (AF) API. These include the following categories of structure:

- Descriptor structures - see Section 8.2.1
- Event structures - see Section 8.2.2
- Other structures - see Section 8.2.3

Descriptor structures These structures are used to represent the following descriptors that contain information about the host node:

- Node descriptor
- Node Power descriptor
- Simple descriptor

The structures are listed below.

1. ZPS_tsAplAfNodeDescriptor
2. ZPS_tsAplAfNodePowerDescriptor
3. ZPS_tsAplAfSimpleDescriptor

ZPS_tsAplAfNodeDescriptor The AF Node descriptor structure ZPS_tsAplAfNodeDescriptoris shown below.

```
typedef struct {
    uint32 :                    8;/* padding */
    uint32 eLogicalType :       3;
    uint32 bComplexDescAvail :  1;
    uint32 bUserDescAvail :     1;
    uint32 eReserved :          3; /* reserved */
    uint32 eFrequencyBand :     5;
    uint32 eApsFlags :          3;
    uint32 u8MacFlags :         8;
    uint16 u16ManufacturerCode;
    uint8 u8MaxBufferSize;
    uint16 u16MaxRxSize;
    uint16 u16ServerMask;
    uint16 u16MaxTxSize;
    uint8 u8DescriptorCapability;
} ZPS_tsAplAfNodeDescriptor;
```

where:

- eLogicalTypecontains 3 bits (bits 0-2) indicating the ZigBee device type of the node, as follows:
  - 000: Coordinator
  - 001: Router
  - 010: End Device
- bComplexDescAvailis set to 1 if there is a Complex descriptor available for node.
- bUserDescAvailis set to 1 if there is a User descriptor available for node.
- eReservedis reserved.
- eFrequencyBandcontains 5 bits detailing the frequency bands supported by the node, as follows (a bit is set to 1 if the corresponding band is supported):
  - Bit 0: 868-868.6 MHz
  - Bit 2: 902-928 MHz
  - Bit 3: 2400-2483.5 MHz
  - Bits 1 and 4 are reserved
- eApsFlagsis not currently supported and set to zero.
- eMacFlagscontains 8 bits (bits 0-7) indicating the node capabilities, as required by the IEEE 802.15.4 MAC sub-layer. These node capability flags are described in the table: Table.
- u16ManufacturerCodecontains 16 bits (bits 0-15) indicating the manufacturer code for the node, where this code is allocated to the manufacturer by the ZigBee Alliance.
- u8MaxBufferSizeis the maximum size, in bytes, of an NPDU (Network Protocol Data Unit).
- u16MaxRxSizeis the maximum size, in bytes, of an APDU (Application Protocol Data Unit). This value can be greater than the value of u8MaxBufferSize, due to the fragmentation of an APDU into NPDUs.

- u16ServerMaskcontains 8 bits (bits 0-7) indicating the server status of the node. This server mask is detailed in the table: Table.
- u16MaxTxSizeis the maximum size, in bytes, of the ASDU (Application Sub-layer Data Unit) in which a message can be sent (the message may actually be transmitted in smaller fragments)
- u8DescriptorCapabilitycontains 8 bits (bits 0-7) indicating the properties of the node that can be used by other nodes in network discovery, as follows:

Bit description of u8DescriptorCapability

Bit	Description
0	Set to 1 if Extended Active Endpoint List is available on the node, 0 otherwise.
1	Set to 1 if Extended Simple Descriptor List is available on the node, 0 otherwise.
2-7	Reserved

Parent topic: Descriptor structures

ZPS_tsAplAfNodePowerDescriptor The AF Node Power descriptor structure ZPS_tsAplAfNodePowerDescriptoris shown below.

```
typedef struct {
    uint32 eCurrentPowerMode          : 4;
    uint32 eAvailablePowerSources     : 4;
    uint32 eCurrentPowerSource        : 4;
    uint32 eCurrentPowerSourceLevel   : 4;
} ZPS_tsAplAfNodePowerDescriptor;
```

where:

- eCurrentPowerModecontains 4 bits (bits 0-3) indicating the power mode currently used by the node, as follows:
  - 0000: Receiver configured according to “Receiver on when idle” MAC flag in the Node Descriptor (see Section 8.2.1.1)
  - 0001: Receiver switched on periodically
  - 0010: Receiver switched on when stimulated, for example, by pressing a button
  - All other values are reserved
- eAvailablePowerSourcescontains 4 bits (bits 0-3) indicating the available power sources for the node, as follows (a bit is set to 1 if the corresponding power source is available):
  - Bit 0: Permanent mains supply
  - Bit 1: Rechargeable battery
  - Bit 2: Disposable battery
  - Bit 4: Reserved
- eCurrentPowerSourcecontains 4 bits (bits 0-3) indicating the current power source for the node, as detailed for the element above (the bit corresponding to the current power source is set to 1, all other bits are set to 0).
- eCurrentPowerSourceLevelcontains 4 bits (bit 0-3) indicating the current level of charge of the node’s power source (mainly useful for batteries), as follows:
  - 0000: Critically low
  - 0100: Approximately 33%
  - 1000: Approximately 66%

– 1100: Approximately 100% (near fully charged)

Parent topic: Descriptor structures

ZPS_tsAplAfSimpleDescriptor The AF Simple descriptor structure ZPS_tsAplAfSimpleDescriptoris shown below.

```
typedef struct {
    uint16 u16ApplicationProfileId;
    uint16 u16DeviceId;
    uint8 u8DeviceVersion;
    uint8 u8Endpoint;
    uint8 u8InClusterCount;
    uint8 u8OutClusterCount;
    uint16 *pu16InClusterList;
    uint16 *pu16OutClusterList;
} ZPS_tsAplAfSimpleDescriptor;
```

where:

- u16ApplicationProfileIdis the 16-bit identifier of the ZigBee application profile supported by the endpoint. This must be an application profile identifier issued by the ZigBee Alliance (for Lighting and Occupancy devices, it is 0x0104).
- u16DeviceIdis the 16-bit identifier of the ZigBee device type supported by the endpoint. This must be a device type identifier issued by the ZigBee Alliance.
- u8DeviceVersioncontains 4 bits (bits 0-3) representing the version of the supported device description (default is 0000, unless set to another value according to the application profile used).
- u8Endpointis the number, in the range 1-240, of the endpoint to which the Simple descriptor corresponds.
- u8InClusterCountis an 8-bit count of the number of input clusters, supported on the endpoint, that will appear in the list pointed to by the pu16InClusterListelement.
- u8OutClusterCountis an 8-bit count of the number of output clusters, supported on the endpoint, that will appear in the pu16OutClusterListelement.
- *pu16InClusterListis a pointer to the list of input clusters supported by the endpoint (for use during the service discovery and binding procedures). This is a sequence of 16-bit values, representing the cluster numbers (in the range 1-240), where the number of values is equal to count u8InClusterCount. If this count is zero, the pointer can be set to NULL.
- *pu16OutClusterListis a pointer to the list of output clusters supported by the endpoint (for use during the service discovery and binding procedures). This is a sequence of 16-bit values, representing the cluster numbers (in the range 1-240), where the number of values is equal to count u8OutClusterCount. If this count is zero, the pointer can be set to NULL.

Parent topic: Descriptor structures

Parent topic:AF structures

Event structures These structures are used to contain events. Event details (type and associated data) are passed to the application in the structure ZPS_tsAfEvent. Data structures for the individual event types are contained in the union ZPS_tuAfEventData.

Enumerations for the event types are provided in the structure ZPS_teAfEventType. This structure and the associated events are detailed in Chapter 11, *Event and Status Codes*.

The structures are listed below.

1. ZPS_tsAfEvent

2. ZPS_tuAfEventData
3. ZPS_tsAfDataIndEvent
4. ZPS_tsAfDataConfEvent
5. ZPS_tsAfDataAckEvent
6. ZPS_tsAfNwkFormationEvent
7. ZPS_tsAfNwkJoinedEvent
8. ZPS_tsAfNwkJoinFailedEvent
9. ZPS_tsAfNwkDiscoveryEvent
10. ZPS_tsAfNwkJoinIndEvent
11. ZPS_tsAfNwkLeaveIndEvent
12. ZPS_tsAfNwkLeaveConfEvent
13. ZPS_tsAfNwkStatusIndEvent
14. ZPS_tsAfNwkRouteDiscoveryConfEvent
15. ZPS_tsAfPollConfEvent
16. ZPS_tsAfNwkEdScanConfEvent
17. ZPS_tsAfErrorEvent
18. ZPS_tsAfZdoBindEvent
19. ZPS_tsAfZdoUnbindEvent
20. ZPS_tsAfZdoLinkKeyEvent
21. ZPS_tsAfBindRequestServerEvent
22. ZPS_tsAfInterPanDataIndEvent
23. ZPS_tsAfInterPanDataConfEvent
24. ZPS_tsAfTCstatusEvent
25. ZPS_tsAfZdpEvent

ZPS_tsAfEvent This structure contains the details of an event. The ZPS_tsAfEventstructure is detailed below.

```
typedef struct {
    ZPS_teAfEventType eType;
    ZPS_tuAfEventData uEvent;
} ZPS_tsAfEvent;
```

where

- eTypeindicates the event type, using the enumerations listed and described in Section 11.1.
- uEventis a structure containing the event data from the union of structures detailed in Section 8.2.2.2.

Parent topic: Event structures

ZPS_tuAfEventData This structure is a union of the data structures for the individual events described in Section 8.2.2.3 through to Section 8.2.2.25.

The ZPS_tuAfEventDatastructure is detailed below.

```
typedef union
{
ZPS_tsAfDataIndEvent                sApsDataIndEvent;
ZPS_tsAfDataConfEvent               sApsDataConfirmEvent;
ZPS_tsAfDataAckEvent                sApsDataAckEvent;
ZPS_tsAfNwkFormationEvent           sNwkFormationEvent;
ZPS_tsAfNwkJoinedEvent              sNwkJoinedEvent;
ZPS_tsAfNwkJoinFailedEvent          sNwkJoinFailedEvent;
ZPS_tsAfNwkDiscoveryEvent           sNwkDiscoveryEvent;
ZPS_tsAfNwkJoinIndEvent             sNwkJoinIndicationEvent;
ZPS_tsAfNwkLeaveIndEvent            sNwkLeaveIndicationEvent;
ZPS_tsAfNwkLeaveConfEvent           sNwkLeaveConfirmEvent;
ZPS_tsAfNwkStatusIndEvent           sNwkStatusIndicationEvent;
ZPS_tsAfNwkRouteDiscoveryConfEvent  sNwkRouteDiscoveryConfirmEvent;
ZPS_tsAfPollConfEvent               sNwkPollConfirmEvent;
ZPS_tsAfNwkEdScanConfEvent          sNwkEdScanConfirmEvent;
ZPS_tsAfErrorEvent                  sAfErrorEvent;
ZPS_tsAfZdoBindEvent                sZdoBindEvent;
ZPS_tsAfZdoUnbindEvent              sZdoUnbindEvent;
ZPS_tsAfZdoLinkKeyEvent             sZdoLinkKeyEvent;
ZPS_tsAfBindRequestServerEvent      sBindRequestServerEvent;
ZPS_tsAfInterPanDataIndEvent        sApsInterPanDataIndEvent;
ZPS_tsAfInterPanDataConfEvent       sApsInterPanDataConfirmEvent;
ZPS_tsAfZdpEvent                    sApsZdpEvent;
} ZPS_tuAfEventData;
```

Parent topic: Event structures

ZPS_tsAfDataIndEvent This structure is used in the ZPS_EVENT_APS_DATA_INDICATION event, which indicates the arrival of data on the local node.

The ZPS_tsAfDataIndEventstructure is detailed below.

```
typedef struct
{
uint8           u8DstAddrMode;
ZPS_tuAddress   uDstAddress;
uint8           u8DstEndpoint;
uint8           u8SrcAddrMode;
ZPS_tuAddress   uSrcAddress;
uint8           u8SrcEndpoint;
uint16          u16ProfileId;
uint16          u16ClusterId;
PDUM_thAPduInstance hAPduInst;
uint8           eStatus;
uint8           eSecurityStatus;
uint8           u8LinkQuality;
uint32          u32RxTime;
} ZPS_tsAfDataIndEvent;
```

where:

- u8DstAddrModeindicates the type of destination address specified through the element uDstAddress(see the Table below.)
- uDstAddressis the address of the destination node for the data packet (the type of address is specified using the element u8DstAddrModeabove).
- u8DstEndpointis the number of the destination endpoint (in range 0-240).
- u8SrcAddrModeindicates the type of source address specified through the element uSrcAddress(below) - this can be a 64-bit MAC/IEEE address or a 16-bit network address.

- uSrcAddressis the address of the source node for the data packet (the type of address is specified using the element u8SrcAddrModeabove).
- u8SrcEndpointis the number of the source endpoint (in range 1-240).
- u16ProfileIdis the identifier of the ZigBee device profile of the device which can interpret the data.
- u16ClusterIdis the identifier of the cluster (which belongs to the device profile specified in u16ProfileId) which is capable of interpreting the data.
- hAPduInstis the handle of the APDU which contains the data.
- eStatusis one of the status codes from the NWK layer or MAC layer, detailed in Section 11.2.3 and Section 11.2.4.
- eSecurityStatusindicates the type of security with which the packet was sent. It can be: unsecured (0xAF), secured with network key (0xAC), or secured with link key (0xAB).
- u8LinkQualityis a measure of the signal strength of the radio link over which the data packet was sent (for the last hop).
- u32RxTimeis reserved for future use.

Addressing modes

u8DstAddrMode	Code	Description
0x00	ZPS_E_ADDR_MODE_BOUND	Bound endpoint
0x01	ZPS_E_ADDR_MODE_GROUP	16-bit Group address
0x02	ZPS_E_ADDR_MODE_SHORT	16-bit Network (Short) address
0x03	ZPS_E_ADDR_MODE_IEEE	64-bit IEEE/MAC address

Parent topic: Event structures

ZPS_tsAfDataConfEvent This structure is used in the ZPS_EVENT_APS_DATA_CONFIRM event, which confirms that a data packet sent by the local node has been successfully passed down the stack to the MAC layer and has made its first hop toward its destination (an acknowledgment has been received from the next hop node).

The ZPS_tsAfDataConfEventstructure is detailed below.

```
typedef struct {
uint8        u8Status;
uint8        u8SrcEndpoint;
uint8        u8DstEndpoint;
uint8        u8DstAddrMode;
ZPS_tuAddress uDstAddr;
uint8        u8SequenceNum;
} ZPS_tsAfDataConfEvent;
```

where:

- u8Statusis one of the status codes from the lower stack layers, detailed in Section 11.2.
- u8SrcEndpointis the number of the (local) source endpoint for the data transfer (in range 1-240).
- u8DstEndpointis the number of the destination endpoint for the data transfer (in range 1-240).
- u8DstAddrModeindicates the type of destination address specified through the element uDstAddr(see Table) - only values 0x02 (group address) and 0x03 (network address) are valid in this structure.

- uDstAddris the address of the destination node for the data packet (the type of address is specified using the element u8DstAddrModeabove).
- u8SequenceNumis the sequence number of the request that initiated the data transfer.

Parent topic: Event structures

ZPS_tsAfDataAckEvent This structure is used in the ZPS_EVENT_APS_DATA_ACK event, which is generated when an end-to-end acknowledgment is received from the destination node during a data transfer in which an acknowledgment was requested.

```
typedef struct {
  uint8 u8Status;
  uint8 u8SrcEndpoint;
  uint8 u8DstEndpoint;
  uint8 u8DstAddrMode;
  uint16 u16DstAddr;
  uint8 u8SequenceNum;
  uint16 u16ProfileId;
  uint16 u16ClusterId;
} ZPS_tsAfDataAckEvent;
```

where:

- u8Statusis one of the status codes from the lower stack layers, detailed in Section 11.2.
- u8SrcEndpointis the number of the (local) source endpoint for the data transfer (in range 1-240).
- u8DstEndpointis the number of the destination endpoint for the data transfer (in range 1-240).
- u8DstAddrModeindicates the type of destination address specified through the element u16DstAddr(see Table) - only values 0x01 (group address) and 0x02 (network address) are valid in this structure.
- u16DstAddris the 16-bit address of the destination node for the data transfer and therefore of the node that sent the acknowledgment (the type of address is specified using the element u8DstAddrModeabove).
- u8SequenceNumis the sequence number of the request that initiated the data transfer.
- u16ProfileIdis the identifier of the ZigBee device profile of the device for which the data transfer was intended.
- u16ClusterIdis the identifier of the cluster (which belongs to the device profile specified in u16ProfileId) for which the data transfer was intended.

Parent topic: Event structures

ZPS_tsAfNwkFormationEvent This structure is used in the event ZPS_EVENT_NWK_STARTED, which indicates whether the network has been started (on the Coordinator).

The ZPS_tsAfNwkFormationEventstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAfNwkFormationEvent;
```

where is one of the status codes from the lower stack layers, detailed in Section 11.2.

Parent topic: Event structures

ZPS_tsAfNwkJoinedEvent This structure is used in the events ZPS_EVENT_NWK_JOINED_AS_ROUTER and ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE, which confirm that the local device (Router or End Device) has successfully joined a network.

The ZPS_tsAfNwkJoinedEventstructure reports the network address that the parent has assigned to the new node and is detailed below.

```
typedef struct
{
    uint16 u16Addr;
    bool_t bRejoin;
} ZPS_tsAfNwkJoinedEvent;
```

where:

- u16Addris the 16-bit network address allocated to the joining node.
- bRejoinindicates whether the join was a rejoin (TRUE) or a new association (FALSE).

Parent topic: Event structures

ZPS_tsAfNwkJoinFailedEvent This structure is used in the event ZPS_EVENT_NWK_FAILED_TO_JOIN, which indicates that the local device has failed to join a network.

The ZPS_tsAfNwkJoinFailedEventstructure is detailed below.

```
typedef struct
{
    uint8 u8Status;
    bool_t bRejoin;
} ZPS_tsAfNwkJoinFailedEvent;
```

where:

- u8Statusis one of the status codes from the lower stack layers, detailed in Section 11.2.
- bRejoinindicates whether the join attempt was a rejoin (TRUE) or a new association (FALSE).

Parent topic: Event structures

ZPS_tsAfNwkDiscoveryEvent This structure is used in the ZPS_EVENT_NWK_DISCOVERY_COMPLETE event, which reports the details of the networks detected in a network discovery initiated by a Router or End Device that needs to join a network.

The ZPS_tsAfNwkDiscoveryEventstructure is detailed below.

```
typedef struct
{
    uint32        u32UnscannedChannels;
    uint8         eStatus;
    uint8         u8NetworkCount;
    uint8         u8SelectedNetwork;
    ZPS_tsNwkNetworkDescr *psNwkDescriptors;
} ZPS_tsAfNwkDiscoveryEvent;
```

where:

- u32UnscannedChannelsis a 32-bit bitmap representing the set of channels from the network discovery that had not yet been scanned when this event was generated. Bits 11 to 26 represent the 2400-MHz channels 11 to 26, where 1 indicates channel scanned and 0 indicates channel not yet scanned.

- estatusis the status of the network discovery process, returned by the lower layers (see Section 11.2) - MAC_ENUM_SUCCESS, if the discovery was successfully completed.
- u8NetworkCountis the number of networks that had been discovered when this event was generated.
- u8SelectedNetworkis the index of the recommended network in the array of reported networks (see below).
- psNwkDescriptorsis a pointer to the network discovery table in the network NIB. The network discovery table contains an array of data structures, where each structure contains details of a discovered network. Each array element is a structure of the type ZPS_tsNwkNetworkDescr, described in Section 8.2.3.1. The number of array elements is given by u8NetworkCount, described above.

Parent topic: Event structures

ZPS_tsAfNwkJoinIndEvent This structure is used in the event ZPS_EVENT_NWK_NEW_NODE_HAS_JOINED, which notifies a Router or the Coordinator that a new child node has joined the network.

The ZPS_tsAfNwkJoinIndEventstructure contains information about the new node and is detailed below.

```
typedef struct
{
  uint64 u64ExtAddr;
  uint16 u16NwkAddr;
  uint8 u8Capability;
  uint8 u8Rejoin;
  uint8 u8SecureRejoin;
} ZPS_tsAfNwkJoinIndEvent;
```

where:

- u64ExtAddris the 64-bit IEEE (MAC) address of the joining node.
- u16NwkAddris the 16-bit network address assigned to the joining node.
- u8Capabilityis a bitmap indicating the operational capabilities of the joining node. This bitmap is detailed in Table below.
- u8Rejoinindicates the method used to join the network:
  - 0x00 if joined through association.
  - 0x01 if joined directly or used orphaning.
  - 0x02 if was network rejoin.
- u8SecureRejoinindicates whether the join was performed in a secure manner.
  - zero represents FALSE.
  - a non-zero value represents TRUE.

Node capabilities bitmap

Bits	Description
0	Coordinator capability:- 1: Node able to act as Coordinator- 0: Node not able to act as Coordinator
1	Device type:- 1: Full-Function Device (FFD)- 0: Reduced-Function Device (RFD)An FFD can act as any node type while an RFD cannot act as the network Coordinator.
2	Power source:- 1:Node is mains-powered - 0: Node is not mains-powered
3	Receiver on when idle:- 1: Receiver enabled during idle periods- 0: Receiver disabled during idle periods to conserve power
4-5	Reserved
6	Security capability:- 1: High security- 0: Standard security
7	Allocate address:- 1: Network address should be allocated to node- 0: Network address need not be allocated to node

Parent topic: Event structures

ZPS_tsAfNwkLeaveIndEvent This structure is used in the ZPS_EVENT_LEAVE_INDICATION event, which indicates that a neighboring node has left the network or a remote node has requested the local node to leave.

The ZPS_tsAfNwkLeaveIndEventstructure is detailed below.

```
typedef struct {
    uint64 u64ExtAddr;
    uint8 u8Rejoin;
} ZPS_tsAfNwkLeaveIndEvent;
```

where:

- u64ExtAddris the 64-bit IEEE (MAC) address of the node that has left the network, or is zero if the local node has been requested to leave the network
- u8Rejoinindicates whether the leaving node was requested to attempt a subsequent rejoin of the network:
  - zero represents FALSE
  - a non-zero value represents TRUE.

Parent topic: Event structures

ZPS_tsAfNwkLeaveConfEvent This structure is used in the event ZPS_EVENT_NWK_LEAVE_CONFIRM, which reports the results of a node leave request issued by the local node.

The ZPS_tsAfNwkLeaveConfEventstructure is detailed below.

```
typedef struct {
    uint64 u64ExtAddr;
    uint8 eStatus;
} ZPS_tsAfNwkLeaveConfEvent;
```

where:

- u64ExtAddris the 64-bit IEEE (MAC) address of the leaving node. This value is zero if the local node itself is leaving.
- eStatusis the leave status returned by the lower layers - ZPS_NWK_ENUM_SUCCESS, if the leave request has been successful.

Parent topic: Event structures

ZPS_tsAfNwkStatusIndEvent This structure is used in the ZPS_EVENT_NWK_STATUS_INDICATION event, which reports status information from the NWK layer of the stack.

The ZPS_tsAfNwkStatusIndEventstructure is detailed below.

```
typedef struct {
    uint16 u16NwkAddr;
    uint8 u8Status;
} ZPS_tsAfNwkStatusIndEvent;
```

where:

- u16NwkAddris the 16-bit network address of the node associated with the event.
- u8Statusis one of the status codes from the lower stack layers, detailed in Section 11.2.

Parent topic: Event structures

ZPS_tsAfNwkRouteDiscoveryConfEvent This structure is used in the ZPS_EVENT_NWK_ROUTE_DISCOVERY_CONFIRM event, which confirms that a route discovery has been performed.

The ZPS_tsAfNwkRouteDiscoveryConfEventstructure is detailed below.

```
typedef struct {
    uint16 u16DstAddress;
    uint8 u8Status;
    uint8 u8NwkStatus;
} ZPS_tsAfNwkRouteDiscoveryConfEvent;
```

where:

- u16DstAddressis the destination address for which the route discovery confirm event was generated.
- u8Statusis one of the status codes from the MAC layer, detailed in Section11.2.4.
- u8NwkStatusis one of the status codes from the NWK layer, detailed in Section 11.2.3.

Parent topic: Event structures

ZPS_tsAfPollConfEvent This structure is used in the ZPS_EVENT_NWK_POLL_CONFIRM event, which reports the completion of a poll request sent from the (local) End Device to its parent.

The ZPS_tsAfPollConfEventstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAfPollConfEvent;
```

where u8Statusis one of the status codes from the lower stack layers, detailed in Section 11.2.

Parent topic: Event structures

ZPS_tsAfNwkEdScanConfEvent This structure is used in the ZPS_EVENT_NWK_ED_SCAN event, which indicates that an 'energy detect' scan in the 2.4-GHz radio band is completed.

The ZPS_tsAfNwkEdScanConfEventstructure is defined as:

```
typedef ZPS_tsNwkNlmeCfmEdScan ZPS_tsAfNwkEdScanConfEvent;
```

where ZPS_tsNwkNlmeCfmEdScanis described in Section 8.2.3.2.

Parent topic: Event structures

ZPS_tsAfErrorEvent This structure is used in the ZPS_EVENT_ERROR event, which reports error situations concerning the storage of received messages in APDU instances.

The ZPS_tsAfErrorEventstructure is detailed below.

```
typedef struct {
    enum {
        ZPS_ERROR_APDU_TOO_SMALL,
        ZPS_ERROR_APDU_INSTANCES_EXHAUSTED,
        ZPS_ERROR_NO_APDU_CONFIGURED,
        ZPS_ERROR_OS_MESSAGE_QUEUE_OVERRUN
    } eError;
   union {
        struct {
            uint16 u16ProfileId;
            uint16 u16ClusterId;
            uint16 u16SrcAddr;
            uint16 u16DataSize;
            PDUM_thAPdu hAPdu;
            uint8 u8SrcEndpoint;
            uint8 u8DstEndpoint;
        }sAfErrorApdu;
        struct {
            OS_thMessage hMessage;
        } sAfErrorOsMessageOverrun;
    } uErrorData;
} ZPS_tsAfErrorEvent;
```

The member enumerations and structures of the above structure are detailed below.

eError enumerations The error enumerations which are part of the ZPS_tsAfErrorEventstructure are listed and described below.

eError Enumerations

eError Enumeration	Description
ZPS_ERROR_APDU_TC	Allocated APDU instance is too small to accommodate received message. This error is detailed in the structure sAfErrorApdu, which is described below.
ZPS_ERROR_APDU_IN	The are no APDU instances available to accommodate the received message. This error is detailed in the structure sAfErrorApdu, which is described below.
ZPS_ERROR_NO_APDU	No APDU has been configured to accommodate the received message. This error is detailed in the structure sAfErrorApdu, which is described below.
ZPS_ERROR_OS_MESS	A message queue is full and can accept no more messages. This error is detailed in the structure sAfErrorOsMessageOverrun, which is described below.

sAfErrorApdu

This structure is used in the following errors:

- ZPS_ERROR_APDU_TOO_SMALL, which reports that the allocated APDU instance is too small to store a received message.
- ZPS_ERROR_APDU_INSTANCES_EXHAUSTED, which reports that there are no allocated APDU instances left to store a received message.

- ZPS_ERROR_NO_APDU_CONFIGURED, which reports that no APDU has been configured to store the received message.

The sAfErrorApdustructure is detailed below.

```
struct {
    uint16 u16ProfileId;
    uint16 u16ClusterId;
    uint16 u16SrcAddr;
    uint16 u16DataSize;
    PDUM_thAPdu hAPdu;
    uint8 u8SrcEndpoint;
    uint8 u8DstEndpoint;
}sAfErrorApdu;
```

where:

- u16ProfileIdis the identifier of the ZigBee application profile associated with the source and destination endpoints for the message.
- u16ClusterIdis the identifier of the cluster associated with the source and destination endpoints for the message.
- u16SrcAddris the 16-bit network address of the source node of the message.
- u16DataSizeis the size of the received message, in bytes.
- hAPduis the handle of the local APDU pool from which the APDU instance comes.
- u8SrcEndpointis the number of the source endpoint of the message.
- u8DstEndpointis the number of the destination endpoint of the message.

Parent topic:ZPS_tsAfErrorEvent

sAfErrorOsMessageOverrun This structure is used in the ZPS_ERROR_OS_MESSAGE_QUEUE_OVERRUN error, which indicates that a message queue is full and can accept no more messages.

The sAfErrorOsMessageOverrunstructure is detailed below.

```
struct {
    OS_thMessage hMessage;
} sAfErrorOsMessageOverrun;
```

where hMessageis the handle of the message type for the queue which is full.

Parent topic:ZPS_tsAfErrorEvent

Parent topic: Event structures

ZPS_tsAfZdoBindEvent This structure is used in the ZPS_EVENT_ZDO_BIND event, which indicates that the local node has been successfully bound to one or more remote nodes.

The ZPS_tsAfZdoBindEventstructure is detailed below.

```
typedef struct { ZPS_tuAddress uDstAddr; uint8 u8DstAddrMode; uint8 u8SrcEp; uint8 u8DstEp; } ZPS_
↪tsAfZdoBindEvent;
```

where

- uDstAddris the address of the remote node for the binding (the type of address is specified using the element u8DstAddrModeabove).
- u8DstAddrModeindicates the type of address specified through the element uDstAddr(see Table 1).

- u8SrcEpis the number of the source endpoint for the binding (in range 1-240).
- u8DstEpis the number of the destination endpoint for the binding (in range 1-240).

Parent topic: Event structures

ZPS_tsAfZdoUnbindEvent This structure is used in the ZPS_EVENT_ZDO_UNBIND event, which indicates that the local node has been successfully unbound from one or more remote nodes.

The ZPS_tsAfZdoUnbindEventstructure is defined as:

```
typedef ZPS_tsAfZdoBindEvent ZPS_tsAfZdoUnbindEvent;
```

where ZPS_tsAfZdoBindEventis described in Section 8.2.2.18\ (but for this event, the data in the structure relates to unbinding rather than binding).

Parent topic: Event structures

ZPS_tsAfZdoLinkKeyEvent This structure is used in the ZPS_EVENT_ZDO_LINK_KEY event, which indicates that a new application link key has been received and installed, and is ready for use.

The ZPS_tsAfZdoLinkKeyEventstructure is defined as:

```
typedef struct {
    uint64 u64IeeeLinkAddr;
} ZPS_tsAfZdoLinkKeyEvent;
```

where u64IeeeLinkAddris the IEEE/MAC address of the remote device with which the installed link key is valid.

Parent topic: Event structures

ZPS_tsAfBindRequestServerEvent This structure is used in the ZPS_EVENT_BIND_REQUEST_SERVER event, which reports the status of a data transmission sent from the (local) node to a set of bound endpoints.

The ZPS_tsAfBindRequestServerEventstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint8 u8SrcEndpoint;
    uint32 u32FailureCount;
} ZPS_tsAfBindRequestServerEvent;
```

where:

- u8Statusis the overall status of the bound data transmission:
  - Success (0) indicates that the data packet was successfully transmitted to all bound endpoints
  - Failure (non-zero value) indicates that the data packet was not successfully sent to at least one bound endpoint (see u32FailureCountbelow).
- u8SrcEndpointis the number of the local endpoint from which the data packet was sent.
- u32FailureCountis the number of bound endpoints for which the transmission failed.

Parent topic: Event structures

ZPS_tsAfInterPanDataIndEvent This structure is used in the ZPS_EVENT_APS_INTERPAN_DATA_INDICATION event, which indicates that an inter-PAN data packet has arrived.

The ZPS_tsAfInterPanDataIndEventstructure is detailed below.

```
typedef struct
{
    ZPS_tsInterPanAddress sDstAddr;
    uint8    u8SrcAddrMode;
    uint16   u16SrcPan;
    uint64   u64SrcAddress;
    uint16   u16ProfileId;
    uint16   u16ClusterId;
    PDUM_thAPduInstance hAPduInst;
    uint8    eStatus;
    uint8    u8DstEndpoint;
    uint8    u8LinkQuality;
} ZPS_tsAfInterPanDataIndEvent;
```

where

- sDstAddris a structure of the type ZPS_tsInterPanAddress(see Section8.2.3.3) which contains the PAN ID and address for the destination node(s) of the inter-PAN data packet.
- u8SrcAddrModeindicates the type of address specified through the element u64SrcAddress (see Table).
- u16SrcPanis the PAN ID of the network from which the data packet originates.
- u64SrcAddressis the address of the node which sent the data packet (the type of address is specified using the element u8SrcAddrModeabove).
- u16ProfileIdis the identifier of the application profile for which the data packet is intended.
- u16ClusterIdis the identifier of the cluster for which the data packet is intended.
- hAPduInstis the handle of the APDU instance for the data packet.
- eStatusis one of the status codes from the lower stack layers, detailed in Section 11.2.
- u8DstEndpointis the number of the destination endpoint for the data packet (in range 1-240).
- u8LinkQualityis an LQI value indicating the perceived strength of the radio signal which carried the received data packet.

Parent topic: Event structures

ZPS_tsAfInterPanDataConfEvent This structure is used in the ZPS_EVENT_APS_INTERPAN_DATA_CONFIRM event, which indicates that an inter-PAN communication has been sent by the local node and an acknowledgment has been received from the first hop node (this acknowledgment is not generated in the case of a broadcast).

The ZPS_tsAfInterPanDataConfEventstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint8 u8Handle;
} ZPS_tsAfInterPanDataConfEvent;
```

where

- u8Statusis one of the status codes from the lower stack layers, detailed in Section 11.2.
- u8Handleis a handle for internal use.

Parent topic: Event structures

ZPS_tsAfTCstatusEvent This structure is used in the ZPS_EVENT_TC_STATUS event, which indicates whether negotiations to establish a link key with the Trust Centre have been successful and, if so, which key is the active key.

The ZPS_tsAfTCstatusEventstructure is detailed below.

```
typedef struct
{
    ZPS_tuTcStatusData  uTcData;
    uint8               u8Status;
}ZPS_tsAfTCstatusEvent;
```

where:

- uTcDatais dependent on u8Status(below) and is either a pointer to the link key descriptor in the case of success or the address of the Trust Centre node in the case of failure. ZPS_tuTcStatusDatais a union, detailed below.
- u8Statusindicates the results of the link key negotiations - one of:
  - ZPS_E_SUCCESS (link key successfully established)
  - ZPS_APL_APS_E_SECURITY_FAIL (link key not established)

The ZPS_tuTcStatusDatastructure is detailed below.

```
typedef union {
   ZPS_tsAplApsKeyDescriptorEntry *pKeyDesc;
   uint64 u64ExtendedAddress;
} PS_tuTcStatusData;
```

where:

- pKeyDescis a pointer to the active link key, if successfully established, which is contained in the structure described in Section 8.2.3.6.
- u64ExtendedAddressis the IEEE/MAC address of the Trust Centre node with which link key negotiations failed.

Parent topic: Event structures

ZPS_tsAfZdpEvent This structure is used when a ZPS_EVENT_APS_DATA_INDICATION event is generated containing a response which is destined for the ZDO at endpoint 0. The application can extract the response data from the event using the function **ZPS_bAplZdpUnpackResponse()** and this structure is used to receive the extracted data.

The ZPS_tsAfZdpEventstructure is detailed below.

```
typedef struct {
    uint8 u8SequNumber;
    uint16 u16ClusterId;
    union {
        ZPS_tsAplZdpDeviceAnnceReq sDeviceAnnce;
        ZPS_tsAplZdpMgmtNwkUpdateReq sMgmtNwkUpdateReq;
        ZPS_tsAplZdpMgmtPermitJoiningReq sPermitJoiningReq;
        ZPS_tsAplZdpDiscoveryCacheRsp sDiscoveryCacheRsp;
        ZPS_tsAplZdpDiscoveryStoreRsp sDiscoveryStoreRsp;
        ZPS_tsAplZdpNodeDescStoreRsp sNodeDescStoreRsp;
        ZPS_tsAplZdpActiveEpStoreRsp sActiveEpStoreRsp;
        ZPS_tsAplZdpSimpleDescStoreRsp sSimpleDescStoreRsp;
        ZPS_tsAplZdpRemoveNodeCacheRsp sRemoveNodeCacheRsp;
        ZPS_tsAplZdpEndDeviceBindRsp sEndDeviceBindRsp;
        ZPS_tsAplZdpBindRsp sBindRsp;
        ZPS_tsAplZdpUnbindRsp sUnbindRsp;
```

(continues on next page)

(continued from previous page)

```
        ZPS_tsAplZdpReplaceDeviceRsp sReplaceDeviceRsp;
        ZPS_tsAplZdpStoreBkupBindEntryRsp sStoreBkupBindEntryRsp;
        ZPS_tsAplZdpRemoveBkupBindEntryRsp sRemoveBkupBindEntryRsp;
        ZPS_tsAplZdpBackupSourceBindRsp sBackupSourceBindRsp;
        ZPS_tsAplZdpMgmtLeaveRsp sMgmtLeaveRsp;
        ZPS_tsAplZdpMgmtDirectJoinRsp sMgmtDirectJoinRsp;
        ZPS_tsAplZdpMgmtPermitJoiningRsp sPermitJoiningRsp;
        ZPS_tsAplZdpNodeDescRsp sNodeDescRsp;
        ZPS_tsAplZdpPowerDescRsp sPowerDescRsp;
        ZPS_tsAplZdpSimpleDescRsp sSimpleDescRsp;
        ZPS_tsAplZdpNwkAddrRsp sNwkAddrRsp;
        ZPS_tsAplZdpIeeeAddrRsp sIeeeAddrRsp;
        ZPS_tsAplZdpUserDescConf sUserDescConf;
        ZPS_tsAplZdpSystemServerDiscoveryRsp sSystemServerDiscoveryRsp;
        ZPS_tsAplZdpPowerDescStoreRsp sPowerDescStoreRsp;
        ZPS_tsAplZdpUserDescRsp sUserDescRsp;
        ZPS_tsAplZdpActiveEpRsp sActiveEpRsp;
        ZPS_tsAplZdpMatchDescRsp sMatchDescRsp;
        ZPS_tsAplZdpComplexDescRsp sComplexDescRsp;
        ZPS_tsAplZdpFindNodeCacheRsp sFindNodeCacheRsp;
        ZPS_tsAplZdpExtendedSimpleDescRsp sExtendedSimpleDescRsp;
        ZPS_tsAplZdpExtendedActiveEpRsp sExtendedActiveEpRsp;
        ZPS_tsAplZdpBindRegisterRsp sBindRegisterRsp;
        ZPS_tsAplZdpBackupBindTableRsp sBackupBindTableRsp;
        ZPS_tsAplZdpRecoverBindTableRsp sRecoverBindTableRsp;
        ZPS_tsAplZdpRecoverSourceBindRsp sRecoverSourceBindRsp;
        ZPS_tsAplZdpMgmtNwkDiscRsp sMgmtNwkDiscRsp;
        ZPS_tsAplZdpMgmtLqiRsp sMgmtLqiRsp;
        ZPS_tsAplZdpMgmtRtgRsp sRtgRsp;
        ZPS_tsAplZdpMgmtBindRsp sMgmtBindRsp;
        ZPS_tsAplZdpMgmtCacheRsp sMgmtCacheRsp;
        ZPS_tsAplZdpMgmtNwkUpdateNotify sMgmtNwkUpdateNotify;
    }uZdpData;
    union {
        ZPS_tsAplZdpBindingTableEntry asBindingTable[5];
        ZPS_tsAplZdpNetworkDescr asNwkDescTable[5];
        ZPS_tsAplZdpNtListEntry asNtList[2];
        ZPS_tsAplDiscoveryCache aDiscCache[5];
        uint16 au16Data[34];
        uint8 au8Data[77];
        uint64 au64Data[9];
    }uLists;
}ZPS_tsAfZdpEvent;
```

where:

- u8SequNumberis the sequence number of the ZDP request/response
- u16ClusterIdis the ID of the cluster to which the request/response relates
- uZdpDatais a union of the different ZDP request/response types:
  - sDeviceAnnceis a structure of the type ZPS_tsAplZdpDeviceAnnceReq, described in Section 9.2.2.3
- sMgmtNwkUpdateReqis a structure of the type ZPS_tsAplZdpMgmtNwkUpdateReq, described in Section 9.2.2.41
- sPermitJoiningReqis a structure of the type ZPS_tsAplZdpMgmtPermitJoiningReq, described in Section 9.2.3.39
- sDiscoveryCacheRspis a structure of the type ZPS_tsAplZdpDiscoveryCacheRsp, described in Section 9.2.3.14

- sDiscoveryStoreRspis a structure of the type ZPS_tsAplZdpDiscoveryStoreRsp, described in Section 9.2.3.15
- sNodeDescStoreRspis a structure of the type ZPS_tsAplZdpNodeDescStoreRsp, described in Section 9.2.3.16
- sActiveEpStoreRspis a structure of the type ZPS_tsAplZdpActiveEpStoreRsp, described in Section 9.2.3.19
- sSimpleDescStoreRspis a structure of the type ZPS_tsAplZdpSimpleDescStoreRsp, described in Section 9.2.3.18
- sRemoveNodeCacheRspis a structure of the type ZPS_tsAplZdpRemoveNodeCacheRsp, described in Section 9.2.3.21
- sEndDeviceBindRspis a structure of the type ZPS_tsAplZdpEndDeviceBindRsp, described in Section 9.2.3.22
- sBindRspis a structure of the type ZPS_tsAplZdpBindRsp, described in Section 9.2.3.23
- sUnbindRspis a structure of the type ZPS_tsAplZdpUnbindRsp, described in Section 9.2.3.24
- sReplaceDeviceRspis a structure of the type ZPS_tsAplZdpReplaceDeviceRsp, described in Section 9.2.3.26
- sStoreBkupBindEntryRspis a structure of the type ZPS_tsAplZdpStoreBkupBindEntryRsp, described in Section9.2.2.27
- sRemoveBkupBindEntryRspis a structure of the type ZPS_tsAplZdpRemoveBkupBindEntryRsp, described in Section9.2.2.28
- sBackupSourceBindRspis a structure of the type ZPS_tsAplZdpBackupSourceBindRsp, described in Section 9.2.3.31
- sMgmtLeaveRspis a structure of the type ZPS_tsAplZdpMgmtLeaveRsp, described in Section 9.2.3.37
- sMgmtDirectJoinRspis a structure of the type ZPS_tsAplZdpMgmtDirectJoinRsp, described in Section 9.2.3.38
- sPermitJoiningRspis a structure of the type ZPS_tsAplZdpMgmtPermitJoiningRsp, described in Section 9.2.3.39
- sNodeDescRspis a structure of the type ZPS_tsAplZdpNodeDescRsp, described in Section 8.2.3.3
- sPowerDescRspis a structure of the type ZPS_tsAplZdpPowerDescRsp, described in Section 9.2.3.4
- sSimpleDescRspis a structure of the type ZPS_tsAplZdpSimpleDescRsp, described in Section 9.2.3.5
- sNwkAddrRspis a structure of the type ZPS_tsAplZdpNwkAddrRsp, described in Section 9.2.3.1
- sIeeeAddrRspis a structure of the type ZPS_tsAplZdpIeeeAddrRsp, described in Section 9.2.3.2
- sUserDescConfis a structure of the type ZPS_tsAplZdpUserDescConf, described in Section 9.2.3.12
- sSystemServerDiscoveryRspis a structure of the type ZPS_tsAplZdpSystemServerDiscoveryRsp, described in Section9.2.3.13
- sPowerDescStoreRspis a structure of the type ZPS_tsAplZdpPowerDescStoreRsp, described in Section 9.2.3.17
- sUserDescRspis a structure of the type ZPS_tsAplZdpUserDescRsp, described in Section 9.2.3.8

- sActiveEpRspis a structure of the type ZPS_tsAplZdpActiveEpRsp, described in Section 9.2.3.10
- sMatchDescRspis a structure of the type ZPS_tsAplZdpMatchDescRsp, described in Section 9.2.3.9
- sComplexDescRspis a structure of the type ZPS_tsAplZdpComplexDescRsp, described in Section 9.2.3.7
- sFindNodeCacheRspis a structure of the type ZPS_tsAplZdpFindNodeCacheRsp, described in Section 9.2.3.20
- sExtendedSimpleDescRspis a structure of the type ZPS_tsAplZdpExtendedSimpleDescRsp, described in Section 9.2.3.6
- sExtendedActiveEpRspis a structure of the type ZPS_tsAplZdpExtendedActiveEpRsp, described in Section 9.2.3.11
- sBindRegisterRspis a structure of the type ZPS_tsAplZdpBindRegisterRsp, described in Section 9.2.3.25
- sBackupBindTableRspis a structure of the type ZPS_tsAplZdpBackupBindTableRsp, described in Section 9.2.3.29
- sRecoverBindTableRspis a structure of the type ZPS_tsAplZdpRecoverBindTableRsp, described in Section 9.2.3.30
- sRecoverSourceBindRspis a structure of the type ZPS_tsAplZdpRecoverSourceBindRsp, described in Section 9.2.3.32
- sMgmtNwkDiscRspis a structure of the type ZPS_tsAplZdpMgmtNwkDiscRsp, described in Section 9.2.3.33
- sMgmtLqiRspis a structure of the type ZPS_tsAplZdpMgmtLqiRsp, described in Section 9.2.3.34
- sRtgRspis a structure of the type ZPS_tsAplZdpMgmtRtgRsp, described in Section 9.2.3.35
- sMgmtBindRspis a structure of the type ZPS_tsAplZdpMgmtBindRsp, described in Section 9.2.3.36
- sMgmtCacheRspis a structure of the type ZPS_tsAplZdpMgmtCacheRsp, described in Section 9.2.3.40
- sMgmtNwkUpdateNotifyis a structure of the type ZPS_tsAplZdpMgmtNwkUpdateNotify, described in Section 9.2.3.41
- uListsis a union of the different arrays/tables which act as temporary storage for data elements used by the stack (and are therefore for internal use only)

Parent topic: Event structures

Parent topic:AF structures

Other structures This section describes various structures used by the AF API. The structures are listed below.

1. ZPS_tsNwkNetworkDescr
2. ZPS_tsNwkNlmeCfmEdScan
3. ZPS_tsInterPanAddress
4. ZPS_tsAplApsKeyDescriptorEntry
5. ZPS_tsAftsStartParamsDistributed
6. ZPS_tsAfFlashInfoSet
7. ZPS_TclkDescriptorEntry

ZPS_tsNwkNetworkDescr This structure is used in an array element in the structure ZPS_tsAfNwkDiscoveryEvent, which is created as part of the ZPS_EVENT_NWK_DISCOVERY_COMPLETE event. This event reports the networks detected during a network discovery (see Section 8.2.2.9).

The ZPS_tsNwkNetworkDescrstructure contains information on a detected network and is detailed below.

```
typedef struct
{
    uint64 u64ExtPanId;
    uint8 u8LogicalChan;
    uint8 u8StackProfile;
    uint8 u8ZigBeeVersion;
    uint8 u8PermitJoining;
    uint8 u8RouterCapacity;
    uint8 u8EndDeviceCapacity;
} ZPS_tsNwkNetworkDescr;
```

where:

- u64ExtPanIdis the Extended PAN ID of the discovered network.
- u8LogicalChanis the 2400-MHz channel on which the network was found.
- u8StackProfileis the Stack Profile of the discovered network:
  - 0 - manufacturer-specific
  - 1 - ZigBee
  - 2 - ZigBee PRO
  - other values reserved, and is fixed at 2 for the NXP stack
- u8ZigBeeVersionis the ZigBee version of the discovered network.
- u8PermitJoiningindicates the number of detected nodes with 'permit joining' enabled (and therefore allowing nodes to join the network through them).
- u8RouterCapacityindicates the number of detected nodes that are allowing Routers to join the network through them.
- u8EndDeviceCapacityindicates the number of detected nodes that are allowing End Devices to join the network through them.

Parent topic: Other structures

ZPS_tsNwkNlmeCfmEdScan This structure is used by the structure ZPS_tsAfNwkEdScanConfEvent, which is created as part of the ZPS_EVENT_NWK_ED_SCAN event which reports the results of an 'energy detect' scan in the 2.4-GHz radio band.

The ZPS_tsNwkNlmeCfmEdScantstructure is detailed below.

```
typedef struct
{
    uint8 u8Status;
    uint8 u8ResultListSize;
    uint8 au8EnergyDetect[ZPS_NWK_MAX_ED_RESULTS];
} ZPS_tsNwkNlmeCfmEdScan;
```

where

- u8Statusis one of the status codes from the lower stack layers, detailed in Section 11.2.
- u8ResultListSizeis the number of entries in the results list (see below).

- au8EnergyDetect[]is an array containing the list of results of the energy scan (8-bit values representing the detected energy levels in the channels). There is one array element for each channel scanned, where element 0 is for the first channel scanned, element 1 is for the second channel scanned, etc.

Parent topic: Other structures

ZPS_tsInterPanAddress This structure is used to specify the destination for an inter-PAN transmission. The ZPS_tsInterPanAddressstructure is detailed below.

```
typedef struct
{
    enum {
        ZPS_E_AM_INTERPAN_GROUP = 0x01,
        ZPS_E_AM_INTERPAN_SHORT,
        ZPS_E_AM_INTERPAN_IEEE
    }eMode;
    uint16 u16PanId;
    ZPS_tuAddress uAddress;
} ZPS_tsInterPanAddress;
```

where:

- eModeis used to specify the type of destination address that will be used in the field uAddressbelow. One of the following enumerations must be specified:
  - ZPS_E_AM_INTERPAN_GROUP indicates that a 16-bit group address will be used to specify multiple target nodes in the destination network (the group address must be valid in the destination network)
  - ZPS_E_AM_INTERPAN_SHORT indicates that a 16-bit network/short address will be used to specify a single target node or a broadcast to all nodes in the destination network
  - ZPS_E_AM_INTERPAN_IEEE indicates that a 64-bit IEEE/MAC address will be used to specify a single target node in the destination network
- u16PanIdis the PAN ID of the destination network - a value 0xFFFF can be used to specify a broadcast to all reachable ZigBee PRO networks
- uAddressis the address of the target node(s) in the destination network (the address type must be as specified above in the eModefield) - a value of 0xFFFF can be used to specify a broadcast to all nodes in the destination network(s).

Parent topic: Other structures

ZPS_tsAfProfileDataReq This structure is used to specify the transmission details for a data transmission submitted using the function **ZPS_eAplAfApsdeDataReq()**.

The ZPS_tsAfProfileDataReqstructure is detailed below.

```
typedef struct {
    ZPS_tuAddress              uDstAddr;
    uint16                     u16ClusterId;
    uint16                     u16ProfileId;
    uint8                      u8SrcEp;
    ZPS_teAplApsdeAddressMode eDstAddrMode;
    uint8                      u8DstEp;
    ZPS_teAplAfSecurityMode    eSecurityMode;
    uint8                      u8Radius;
}ZPS_tsAfProfileDataReq;
```

where:

- uDstAddris the address of the destination node for the transmission request (can be 16- or 64-bit, as specified by eDstAddrMode).
- u16ClusterIdis the Cluster ID of the destination cluster.
- u16ProfileIdis the Profile ID of the destination application profile.
- u8SrcEpis the source endpoint number (1-240) on the local node.
- eDstAddrModeis the type of destination address, one of (also see the table, Table):
  - ZPS_E_ADDR_MODE_BOUND (no address needed for bound nodes).
  - ZPS_E_ADDR_MODE_GROUP (16-bit group address).
  - ZPS_E_ADDR_MODE_SHORT (16-bit network address).
  - ZPS_E_ADDR_MODE_IEEE (64-bit IEEE/MAC address).
- u8DstEpis the destination endpoint number (1-240) on the remote node.
- eSecurityModeis the security mode for the data transfer, one of:
  - ZPS_E_APL_AF_UNSECURE (no security enabled)
  - ZPS_E_APL_AF_SECURE (Application-level security using link key and network key)
  - ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key)
  - ZPS_E_APL_AF_SECURE | ZPS_E_APL_AF_EXT_NONCE (Application-level security using link key and network key with the extended NONCE included in the frame)
  - ZPS_E_APL_AF_WILD_PROFILE (May be combined with the above flags using OR operator. Sends the message using the wildcard profile (0xFFFF) instead of the profile in the associated Simple descriptor).
- u8Radiusis the maximum number of hops permitted to the destination node (zero value specifies that default maximum is to be used).

Parent topic: Other structures

tsBeaconFilterType This structure contains the details of a beacon filter that can be introduced using the function **ZPS_bAppAddBeaconFilter()**.

The tsBeaconFilterTypestructure is detailed below.

```
typedef struct
{
    uint64      *pu64ExtendPanIdList;
    uint16      u16Panid;
    uint16      u16FilterMap;
    uint8       u8ListSize;
    uint8       u8Lqi;
    uint8       u8Depth;
} tsBeaconFilterType;
```

where:

- pu64ExtendPanIdListis a pointer to a list of 64-bit Extended PAN IDs (EPIDs) which acts as a blacklist or whitelist of networks, depending on the settings of bits 0 and 1 in the u8FilterMapbitmap:
  - If this is a blacklist, beacons from networks with EPIDs in the list will not be accepted
  - If this is a whitelist, only beacons from networks with EPIDs in the list will be accepted
- u16Panidis a 16-bit PAN ID on which beacons can be filtered
- u8ListSizeis the number of Extended PAN IDs in the list pointed to by

pu64ExtendPanIdList

- u8Lqiis the minimum LQI value (in the range 0 to 255) of an acceptable beacon (any beacon with LQI value less than this minimum will be filtered out) - if required, this field must be enabled through bit 2 in the u8FilterMapbitmap
- u8Depthis the tree depth of the neighbor device. A value of 0x00 indicates that the device is the ZigBee coordinator for the network.
- u16FilterMapis an 16-bit bitmap detailing the filtering requirements, as follows:

Bit	Enumeration	Description
0	BF_BITMAP_	If set, field pu64ExtendPanIdListpoints to a blacklist of networks.
1	BF_BITMAP_ (0x2)	If set, field pu64ExtendPanIdListpoints to a whitelist of networks.
2	BF_BITMAP_ (0x4)	If set, beacons must be filtered according to LQI value using the minimum in field u8Lqi.
3	BF_BITMAP_ (0x8)	If set, beacons from nodes with capacity for End Device children can be accepted.
4	BF_BITMAP_ (0x10)	If set, beacons from nodes with capacity for Router children can be accepted.
5	BF_BITMAP_ (0x20)	If set, beacons from nodes with 'permit join- ing' enabled can be accepted.
6	BF_BITMAP_ (0x40)	If set, beacons from nodes on a network with the PAN ID in u16Panidcan be accepted.
7	-	Reserved.
8	BF_BITMAP_	If set, beacons from nodes on a network with the depth in u8Depth. if it is set to 0xff - filters out any beacon which is not from the coordinator. Any other value of u8Depth, filters out beacons with greater than or equal to u8Depth.

Note: Bits 0 and 1 must not both be set.

Note: After each discovery or rejoin, the flags contained in the u16FilterMap field are cleared, while all other fields of this structure remain intact.

Parent topic: Other structures

ZPS_tsAplApsKeyDescriptorEntry This structure contains a link key for secured communications with another node.

```
typedef struct
{
    uint32      u32OutgoingFrameCounter;
    uint16      u16ExtAddrLkup;
    uint8       au8LinkKey[ZPS_SEC_KEY_LENGTH];
} ZPS_tsAplApsKeyDescriptorEntry;
```

where:

- u32OutgoingFrameCounteris the outgoing frame counter value which is incremented on each transmission to a destination address below.
- u16ExtAddrLkupis the index of the local look-up table entry that contains the IEEE/MAC address of either the Trust Centre or the target node.
- au8LinkKey[]is an array containing the link key.

Parent topic: Other structures

ZPS_tsAftsStartParamsDistributed This structure contains the start parameter values for a node in a distributed security network.

```
typedef struct
{
    uint64      u64ExtPanId;
    uint8       *pu8NwkKey;
    uint16      u16PanId;
    uint16      u16NwkAddr;
    uint8       u8KeyIndex;
    uint8       u8LogicalChannel;
    uint8       u8NwkupdateId;
} ZPS_tsAftsStartParamsDistributed;
```

where:

- u64ExtPanIdis the Extended PAN ID of the distributed security network.
- pu8NwkKeyis a pointer to a location to receive the network key.
- u16PanIdis the PAN ID of the network.
- u16NwkAddris the network address of the local node.
- u8KeyIndexis the sequence number required to identify the network key in the security set.
- u8LogicalChannelis the number of the radio channel on which the network operates.
- u8NwkupdateIdis a unique byte value which is incremented when the network parameters are updated (and is therefore used to determine whether a receiving node has missed an update).

Parent topic: Other structures

ZPS_tsAfFlashInfoSet This structure contains information about the devices Flash memory sector used for the persistent storage of unique link keys on the Trust Centre, as enabled by the function **ZPS_vTcInitFlash()**.

```
typedef struct
{
    uint16    u16SectorSize;
    uint16    u16CredNodesCount;
    uint8     u8SectorSet;
} ZPS_tsAfFlashInfoSet;
```

where:

- u16SectorSizeis the size, in bytes, of the Flash memory sector used to store the link keys.
- u16CredNodesCountis the maximum number of nodes for which link keys can be stored in the Flash memory sector.
- u8SectorSetis the number of the Flash memory sector used for this storage.

Note: Care should be taken that this sector is set greater than the current flash usage of the image you are running. If this clashes with something else (image or user data), it would lead to flash corruption and the behavior might become non-deterministic.

Parent topic: Other structures

ZPS_TclkDescriptorEntry This structure is used on the Trust Centre to hold information in RAM about a link key for a node, where this link key is held in persistent storage in devices Flash memory, as enabled by the function **ZPS_vTcInitFlash()**. If this feature is used, the application must allocate space for an array of these structures in RAM, with one structure for each potential node in the network.

```
typedef struct
{
    uint16    u16CredOffset;
    uint16    u16TclkRetries;
} ZPS_TclkDescriptorEntry;
```

where:

- u16CredOffsetis the offset, in bytes, of the storage location for the node's link key in the relevant Flash memory sector.
- u16TclkRetriesis the number of retries that were attempted to negotiate the link key between the Trust Centre and the node.

Parent topic: Other structures

Parent topic:AF structures

Parent topic:*Application Framework (AF) API*

ZigBee Device Profile (ZDP) API The chapter describes the resources of the ZigBee Device Profile (ZDP) API. This API is concerned with sending network requests (for example, binding requests) and receiving responses. The API is defined in the header file **zps_apl_zdp.h**.

In this chapter:

- Section 9.1 details the ZDP API functions.
- Section 9.2 details the ZDP API structures.
- Section 9.3 describes the broadcast options when sending requests using the ZDP API functions.

ZDP API functions The ZDP API functions are divided into the following categories:

- **Address Discovery**functions, described in Section 9.1.1.
- **Service Discovery**functions, described in Section 9.1.2.
- **Binding**functions, described in Section 9.1.3.
- **Network Management Service**functions, described in Section 9.1.4.
- **Response Data Extraction**function, described in Section 9.1.5.

Common parameters

All the ZDP API functions, except **ZPS_bAplZdpUnpackResponse()**, are concerned with sending out a request and all use a similar set of parameters. These parameters are described below, but more specific information is provided as part of the function descriptions:

- ***hAPdu***: This is the unique handle of the APDU (Application Protocol Data Unit) instance for the request to be sent (see below).
- ***uDstAddr***: This is the IEEE address or network address of the node to which the request is sent (the parameter *bExtAddr*must be set according to the type of address used). For a broadcast, *uDstAddr*must be set to a special address, as described in Section 9.3.
- ***bExtAddr***: This is a Boolean indicating the type of address specified in the parameter *uDstAddr* as a 64-bit IEEE address (TRUE) or 16-bit network address (FALSE).
- ***pu8SeqNumber***: This is a pointer to the sequence number for the request - each request must have a unique sequence number to help determine the order in which requests were sent. On sending a request, the function automatically increments the sequence number for the next request.
- ***u16ProfileId***: This is the identifier of the ZigBee application profile being used.

- ***psZdpNwkAddrReq***: This is a pointer to a structure representing the request. The structure used is dependent on the specific function. The different request structures are detailed in Section 9.2.2.

APDUs for requests and responses

A request generated by this API is sent in an APDU (Application Protocol Data Unit). A local APDU instance for the request must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()**. This function returns a handle for the APDU instance, which is subsequently used in the relevant ZDP API request function. Once the request is successfully sent, the APDU instance is automatically de-allocated by the stack (there is no need for the application to de-allocate it).

Note: If the request is not successfully sent (the send function does not return ZPS_E_SUCCESS), then the APDU instance is not de-allocated automatically. In such cases, the application should de-allocate it using the PDUM function **PDUM_eAPduFreeAPduInstance()**.

When a response is subsequently received, the stack automatically allocates a local APDU instance and includes its handle in the notification event for the response. Once the response has been dealt with, the application must de-allocate the APDU instance using the function **PDUM_eAPduFreeAPduInstance()**.

Address discovery functions The ZDP Address Discovery functions are concerned with obtaining addresses of nodes in the network.

The functions are listed below:

1. ZPS_eAplZdpNwkAddrRequest
2. ZPS_eAplZdpIEEEAddrRequest
3. ZPS_eAplZdpDeviceAnnceRequest

Note: Further addressing functions are provided in the ZDO API and are described in Section 7.1.3.

ZPS_eAplZdpNwkAddrRequest

```
ZPS\__teStatus ZPS\__eAplZdpNwkAddrRequest\(
        PDUM\__thAPduInstance hAPduInst,
        ZPS\__tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS\__tsAplZdpNwkAddrReq *psZdpNwkAddrReq);
```

Description This function requests the 16-bit network address of the node with a particular 64-bit IEEE (MAC) address. The function sends out an NWK_addr_req request, which can be either unicast or broadcast, as follows:

- Unicast to another node, specified through *uDstAddr*, that will 'know' the required network address (this may be the parent of the node of interest or the Coordinator)
- Broadcast to the network, in which case *uDstAddr*must be set to the special network address 0xFFFF (see Section 9.3)

The IEEE address of the node of interest must be specified in the request, represented by the structure below (detailed further in Section 9.2.2.1).

```
typedef struct {
        uint64 u64IeeeAddr;
        uint8 u8RequestType;
```

(continues on next page)

(continued from previous page)

```
    uint8 u8StartIndex;
} ZPS_tsAplZdpNwkAddrReq;
```

The required network address is received in an NWK_addr_resp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpNwkAddrRsp(detailed in Section 9.2.3.1). Note that this response can optionally contain the network addresses of the responding node's neighbors (this option is selected as part of the request through u8RequestType).

Parent topic:ZPS_eAplZdpNwkAddrRequest

Parameters

- ***hAPduInst:*** Handle of APDU instance in which request is sent
- ***uDstAddr:*** Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr:*** Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- **pu8SeqNumber:* Pointer to sequence number of request
- **psZdpNwkAddrReq:* Pointer to request (see above).

Parent topic:ZPS_eAplZdpNwkAddrRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpNwkAddrRequest

Parent topic: Address discovery functions

ZPS_eAplZdpIEEEAddrRequest

```
ZPS\__teStatus ZPS\__eAplZdpIeeeAddrRequest\(
    PDUM\__thAPduInstance hAPduInst,
    ZPS\__tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS\__tsAplZdpIeeeAddrReq *psZdpIeeeAddrReq);
```

Description This function requests the 64-bit IEEE (MAC) address of the node with a particular 16-bit network address. The function sends an IEEE_addr_req request to the relevant node, specified through *uDstAddr*.

The network address of the node of interest must also be specified in the request, represented by the structure below (detailed further in Section 9.2.2.2).

typedef struct {

uint16 u16NwkAddrOfInterest; uint8 u8RequestType;

uint8 u8StartIndex;

} ZPS_tsAplZdpIeeeAddrReq;

The required IEEE address is received in an IEEE_addr_resp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpIeeeAddrRsp(detailed in Section 9.2.3.2). Note that this response can optionally contain the IEEE addresses of the responding node's neighbors (this option is selected as part of the request through u8RequestType).

Parent topic:ZPS_eAplZdpIEEEAddrRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent
- *uDstAddr* Network address of destination node of request (*bExtAddr*must be set to FALSE - see below)
- *bExtAddr* Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
  - * *pu8SeqNumber* Pointer to sequence number of request
  - * *psZdpIeeeAddrReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpIEEEAddrRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpIEEEAddrRequest

Parent topic: Address discovery functions

ZPS_eAplZdpDeviceAnnceRequest

```
ZPS_teStatus ZPS_eAplZdpDeviceAnnceRequest(
    PDUM_thAPduInstance hAPduInst,
    uint8 *pu8SeqNumber,
    ZPS_tsAplZdpDeviceAnnceReq *psZdpDeviceAnnceReq);
```

Description This function is used to notify other nodes that the local node has joined or rejoined the network. The function broadcasts a Device_annce announcement to the network and is normally automatically called by the ZDO when the local node joins or rejoins the network.

The IEEE (MAC) and allocated network addresses as well as the capabilities of the sending node must be specified in the announcement, represented by the structure below (detailed further in Section 9.2.2.3).

```
typedef struct {
    uint16 u16NwkAddr;
    uint64 u64IeeeAddr;
    uint8 u8Capability;
} ZPS_tsAplZdpDeviceAnnceReq;
```

On receiving this announcement, a network node updates any information it holds that relates to the supplied IEEE and network addresses:

- If it already holds the supplied IEEE address, it updates the corresponding network address with the supplied one (if necessary).
- If it already holds the supplied network address but with a different corresponding IEEE address, the latter is marked as not having a valid corresponding network address.

Parent topic:ZPS_eAplZdpDeviceAnnceRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent
- * *pu8SeqNumber* Pointer to sequence number of announcement
- * *psZdpDeviceAnnceReq* Pointer to announcement (see above)

Parent topic:ZPS_eAplZdpDeviceAnnceRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpDeviceAnnceRequest

Parent topic: Address discovery functions

Parent topic:ZDP API functions

Service Discovery functions The ZDP Service Discovery functions are concerned with obtaining information about the nature and capabilities of a network node.

The functions are listed below.

1. ZPS_eAplZdpNodeDescRequest
2. ZPS_eAplZdpPowerDescRequest
3. ZPS_eAplZdpSimpleDescRequest
4. ZPS_eAplZdpExtendedSimpleDescRequest
5. ZPS_eAplZdpComplexDescRequest
6. ZPS_eAplZdpUserDescRequest
7. ZPS_eAplZdpMatchDescRequest
8. ZPS_eAplZdpActiveEpRequest
9. ZPS_eAplZdpExtendedActiveEpRequest
10. ZPS_eAplZdpUserDescSetRequest
11. ZPS_eAplZdpSystemServerDiscoveryRequest
12. ZPS_eAplZdpDiscoveryCacheRequest
13. ZPS_eAplZdpDiscoveryStoreRequest
14. ZPS_eAplZdpNodeDescStoreRequest

15. ZPS_eAplZdpPowerDescStoreRequest
16. ZPS_eAplZdpSimpleDescStoreRequest
17. ZPS_eAplZdpActiveEpStoreRequest
18. ZPS_eAplZdpFindNodeCacheRequest
19. ZPS_eAplZdpRemoveNodeCacheRequest

ZPS_eAplZdpNodeDescRequest

```
ZPS\_teStatus ZPS\_eAplZdpNodeDescRequest\(
    PDUM\_thAPduInstance hAPduInst,
    ZPS\_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS\_tsAplZdpNodeDescReq *psZdpNodeDescReq);
```

Description This function requests the Node descriptor of the node with a particular network address. The function sends a Node_Desc_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest must be specified in the request, which is represented by the structure below (further detailed in Section 9.2.2.4).

```
typedef struct {
    uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpNodeDescReq;
```

The required Node descriptor is received in a Node_Desc_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpNodeDescRsp(detailed in Section 9.2.3.3).

Parent topic:ZPS_eAplZdpNodeDescRequest

Parameters

- ***hAPduInst:*** Handle of APDU instance in which request is sent
- ***uDstAddr:*** Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr:*** Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- **pu8SeqNumber:* Pointer to sequence number of request
- **psZdpNodeDescReq:* Pointer to request (see above).

Parent topic:ZPS_eAplZdpNodeDescRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpNodeDescRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpPowerDescRequest

```
ZPS\_teStatus ZPS\_eAplZdpPowerDescRequest\(
    PDUM\_thAPduInstance hAPduInst,
    ZPS\_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS\_tsAplZdpPowerDescReq *psZdpPowerDescReq);
```

Description This function requests the Power descriptor of the node with a particular network address. The function sends a Power_Desc_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest must be specified in the request, which is represented by the structure below (further detailed in Section 9.2.2.5).

typedef struct {

uint16 u16NwkAddrOfInterest;

} ZPS_tsAplZdpPowerDescReq;

The required Power descriptor is received in a Power_Desc_rsp response. The descriptor should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpPowerDescRsp(detailed in Section 9.2.3.4).

Parent topic:ZPS_eAplZdpPowerDescRequest

Parameters

- ***hAPduInst:*** Handle of APDU instance in which request is sent
- ***uDstAddr****:* Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr****:* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- ****pu8SeqNumber*:** Pointer to sequence number of request
- ****psZdpPowerDescReq****:* Pointer to request (see above)

Parent topic:ZPS_eAplZdpPowerDescRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpPowerDescRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpSimpleDescRequest

```
ZPS\_teStatus ZPS\_eAplZdpSimpleDescRequest\(
    PDUM\_thAPduInstance hAPduInst,
    ZPS\_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS\_tsAplZdpSimpleDescReq *psZdpSimpleDescReq);
```

Description This function requests the Simple descriptor for a specific endpoint on the node with a particular network address. The function sends a Simple_Desc_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest and the relevant endpoint on the node must be specified in the request, which is represented by the structure below (further detailed in Section 9.2.2.6).

typedef struct {

uint16 u16NwkAddrOfInterest; uint8 u8EndPoint;

} ZPS_tsAplZdpSimpleDescReq;

The required Simple descriptor is received in a Simple_Desc_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpSimpleDescRsp(detailed in Section 9.2.3.5).

Parent topic:ZPS_eAplZdpSimpleDescRequest

Parameters

- ***hAPduInst:*** Handle of APDU instance in which request is sent
- ***uDstAddr:*** Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr:*** Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- ****pu8SeqNumber:*** Pointer to sequence number of request
- ****psZdpSimpleDescReq:*** Pointer to request (see above).

Parent topic:ZPS_eAplZdpSimpleDescRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpSimpleDescRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpExtendedSimpleDescRequest

```
ZPS\_teStatus ZPS\_eAplZdpExtendedSimpleDescRequest\(
  PDUM\_thAPduInstance hAPduInst,
  ZPS\_tuAddress uDstAddr,
  bool bExtAddr,
  uint8 *pu8SeqNumber,
  ZPS\_tsAplZdpExtendedSimpleDescReq
      *psZdpExtendedSimpleDescReq);
```

Description This function requests a cluster list for a specific endpoint on the node with a particular network address. The function should be called if the endpoint has more input or output clusters than could be included in the response to **ZPS_eAplZdpSimpleDescRequest()**. The function sends an Extended_Simple_Desc_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest and the relevant endpoint on the node must be specified in the request, which is represented by the structure below (further detailed in Section 9.2.2.7).

typedef struct { uint16 u16NwkAddr; uint8 u8EndPoint;

uint8 u8StartIndex;

} ZPS_tsAplZdpExtendedSimpleDescReq;

This structure allows you to specify the first input/output cluster of interest in the endpoint's input and output cluster lists. Thus, this should normally be the cluster after the last one reported following a call to **ZPS_eAplZdpSimpleDescRequest()**.

The required cluster information is received in a Extended_Simple_Desc_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpExtendedSimpleDescRsp(detailed in Section 9.2.3.6).

Parent topic:ZPS_eAplZdpExtendedSimpleDescRequest

Parameters

- ***hAPduInst**:* Handle of APDU instance in which request is sent
- ***uDstAddr**:* Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr**:* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- ****pu8SeqNumber**:* Pointer to sequence number of request
- ****psZdpExtendedSimpleDescReq**:* Pointer to request (see above)

Parent topic:ZPS_eAplZdpExtendedSimpleDescRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpExtendedSimpleDescRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpComplexDescRequest

```
ZPS\_teStatus ZPS\_eAplZdpComplexDescRequest\(
    PDUM\_thAPduInstance hAPduInst,
    ZPS\_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
ZPS\_tsAplZdpComplexDescReq *psZdpComplexDescReq);
```

Description This function requests the Complex descriptor of the node with a particular network address. The function sends a Complex_Desc_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest must be specified in the request, which is represented by the structure below (further detailed in Section 9.2.2.8).

typedef struct {

uint16 u16NwkAddrOfInterest;

} ZPS_tsAplZdpComplexDescReq;

The required Complex descriptor will be received in a Complex_Desc_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpComplexDescRsp(detailed in Section 9.2.3.7).

Parent topic:ZPS_eAplZdpComplexDescRequest

Parameters

- ***hAPduInst*:** Handle of APDU instance in which request is sent
- ***uDstAddr*:** Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr*:** Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- ****pu8SeqNumber:*** Pointer to sequence number of request
- ****psZdpComplexDescReq*:** Pointer to request (see above)

Parent topic:ZPS_eAplZdpComplexDescRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpComplexDescRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpUserDescRequest

```
ZPS_teStatus ZPS_eAplZdpUserDescRequest(
    PDUM_thAPduInstance hAPduInst,
    ZPS_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
ZPS_tsAplZdpUserDescReq *psZdpUserDescReq);
```

Description This function requests the User descriptor of the node with a particular network address. The function sends a User_Desc_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

Note: This function can only be used to access the User descriptor of a non-NXP device (which supports this descriptor), since the storage of a User descriptor on an NXP remove device is not supported.

The network address of the node of interest must be specified in the request, which is represented by the structure below (further detailed in Section 9.2.2.9).

typedef struct {

uint16 u16NwkAddrOfInterest;

} ZPS_tsAplZdpUserDescReq;

The required User descriptor will be received in a User_Desc_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpUserDescRsp(detailed in Section 9.2.3.8).

Parent topic:ZPS_eAplZdpUserDescRequest

Parameters

- ***hAPduInst:*** Handle of APDU instance in which request is sent
- ***uDstAddr***: Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr***: Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- ****pu8SeqNumber:*** Pointer to sequence number of request
- ****psZdpUserDescReq***: Pointer to request (see above).

Parent topic:ZPS_eAplZdpUserDescRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpUserDescRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpMatchDescRequest

```
ZPS\_teStatus ZPS\_eAplZdpMatchDescRequest\(
    PDUM\_thAPduInstance hAPduInst,
    ZPS\_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS\_tsAplZdpMatchDescReq *psZdpMatchDescReq);
```

Description This function requests responses from network nodes with endpoints that match specified criteria in their Simple descriptors. More specifically, these criteria include: application profile, number of input clusters, number of output clusters, list of input clusters, and list of output clusters. The function sends out a Match_Desc_req command, as a broadcast to all network nodes. It might also be sent as a unicast to either a specific node of interest or to another node that may hold the required information in its primary discovery cache. The wildcard profile (0xFFFF) can be used to match any profile ID.

The request is represented by the structure below (further detailed in Section 9.2.2.10).

```
typedef struct {
    uint16 u16NwkAddrOfInterest;
    uint16 u16ProfileId;
    /* rest of message is variable length */
    uint8 u8NumInClusters;
    uint16* pu16InClusterList;
    uint8 u8NumOutClusters;
    uint16* pu16OutClusterList;
} ZPS_tsAplZdpMatchDescReq;
```

A node with matching endpoint criteria responds with a Match_Desc_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpMatchDescRsp(detailed in Section9.2.3.9).

Parent topic:ZPS_eAplZdpMatchDescRequest

Parameters

- ***hAPduInst:*** Handle of APDU instance in which request is sent
- ***uDstAddr:*** Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr:*** Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- ****pu8SeqNumber:*** Pointer to sequence number of request
- ****psZdpMatchDescReq:*** Pointer to request (see above).

Parent topic:ZPS_eAplZdpMatchDescRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpMatchDescRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpActiveEpRequest

```
ZPS\__teStatus ZPS\__eAplZdpActiveEpRequest\(
    PDUM\__thAPduInstance hAPduInst,
    ZPS\__tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS\__tsAplZdpActiveEpReq *psZdpActiveEpReq);
```

Description This function requests a list of the active endpoints on a remote node. The function sends an Active_EP_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest must be specified in the request, which is represented by the structure below (further detailed in Section 9.2.2.11).

typedef struct {

uint16 u16NwkAddrOfInterest;

} ZPS_tsAplZdpActiveEpReq;

The endpoint list is received in an Active_EP_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpActiveEpRsp(detailed in Section 9.2.3.10).

Parent topic:ZPS_eAplZdpActiveEpRequest

Parameters

- ***hAPduInst*:** Handle of APDU instance in which request is sent
- ***uDstAddr:*** Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr*:** Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- ***pu8SeqNumber*:** Pointer to sequence number of request
- ***psZdpActiveEpReq:*** Pointer to request (see above)

Parent topic:ZPS_eAplZdpActiveEpRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpActiveEpRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpExtendedActiveEpRequest

```
ZPS_teStatus ZPS_eAplZdpExtendedActiveEpRequest(
    PDUM_thAPduInstance hAPduInst,
    ZPS_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS_tsAplZdpExtendedActiveEpReq
    *psZdpExtendedActiveEpReq);
```

Description This function requests a list of the active endpoints on a remote node. The function should be called if the node has more active endpoints than could be included in a response to **ZPS_eAplZdpActiveEpRequest()**. The function sends an Extended_Active_EP_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest must be specified in the request, which is represented by the structure below (further detailed in Section 9.2.2.12).

typedef struct { uint16 u16NwkAddr;

uint8 u8StartIndex;

} ZPS_tsAplZdpExtendedActiveEpReq;

This structure allows you to specify the first endpoint of interest for the request.

The endpoint list is received in an Extended_Active_EP_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpExtendedActiveEpRsp(detailed in Section9.2.3.11).

Parent topic:ZPS_eAplZdpExtendedActiveEpRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpActiveEpReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpExtendedActiveEpRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpExtendedActiveEpRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpUserDescSetRequest

```
ZPS_teStatus ZPS_eAplZdpUserDescSetRequest(
    PDUM_thAPduInstance hAPduInst,
    ZPS_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS_tsAplZdpUserDescSet *psZdpUserDescSetReq);
```

Description This function can be used to configure the User descriptor on a remote node. The function sends a User_Desc_set request either to the remote node or to another node that may hold the relevant User descriptor in its primary discovery cache.

Note: This function can only be used to access the User descriptor of a non-NXP device (which supports this descriptor), since the storage of a User descriptor on an NXP device is not supported.

The network address of the node of interest as well as the required modifications must be specified in the request, which is represented by the structure below (further detailed in Section 9.2.2.13).

typedef struct {

uint16 u16NwkAddrOfInterest; uint8 u8Length;

char szUserDescriptor[ZPS_ZDP_LENGTH_OF_USER_DESC];

} ZPS_tsAplZdpUserDescSet;

If the specified User descriptor was successfully modified, a User_Desc_conf response is received. This response should be collected by the application task using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpUserDescConf(detailed in Section 9.2.3.12).

Parent topic:ZPS_eAplZdpUserDescSetRequest

Parameters

- ***hAPduInst*** Handle of APDU instance in which request is sent
- ***uDstAddr*** Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr:*** Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- ****pu8SeqNumber:*** Pointer to sequence number of request
- ****psZdpUserDescSetReq:*** Pointer to request (see above)

Parent topic:ZPS_eAplZdpUserDescSetRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpUserDescSetRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpSystemServerDiscoveryRequest

```
ZPS\__teStatus ZPS\__eAplZdpSystemServerDiscoveryRequest\(
    PDUM\__thAPduInstance hAPduInst,
    uint8 *pu8SeqNumber,
    ZPS\__tsAplZdpSystemServerDiscoveryReq
                    *psZdpSystemServerDiscoveryReq);
```

Description This function can be used to request information on the available servers hosted by remote nodes (Primary or Backup Trust Centre, Primary or Backup Binding Table Cache, Primary or Backup Discovery Cache, Network Manager). The function broadcasts a System_Server_Discovery_req request to all network nodes.

The required servers must be specified by means of a bitmask in the request, which is represented by the structure below (further detailed in Section 9.2.2.14).

typedef struct {

uint16 u16ServerMask;

} ZPS_tsAplZdpSystemServerDiscoveryReq;

A remote node replies with a System_Server_Discovery_rsp response, indicating which of the requested servers are implemented. This response should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpSystemServerDiscoveryRsp(detailed in Section 9.2.3.13).

Parent topic:ZPS_eAplZdpSystemServerDiscoveryRequest

Parameters

- ***hAPduInst*** Handle of APDU instance in which request is sent.
- ****pu8SeqNumber*** Pointer to sequence number of request
- ****psZdpSystemServerDiscoveryReq*** Pointer to request (see above)

Parent topic:ZPS_eAplZdpSystemServerDiscoveryRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpSystemServerDiscoveryRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpDiscoveryCacheRequest

```
ZPS\__teStatus ZPS\__eAplZdpDiscoveryCacheRequest\(
    PDUM\__thAPduInstance hAPduInst,
    uint8 *pu8SeqNumber,
    ZPS\__tsAplZdpDiscoveryCacheReq
                    *psZdpDiscoveryCacheReq);
```

Description This function is used to discover which nodes in the network have a primary discovery cache - that is, a bank of information about other nodes in the network. The function broadcasts a Discovery_Cache_req request to the network.

The request includes the network and IEEE addresses of the sending device, and is represented by the structure below (further detailed in Section 9.2.2.15).

typedef struct {

uint16 u16NwkAddr;

uint64 u64IeeeAddr;

} ZPS_tsAplZdpDiscoveryCacheReq;

A node with a primary discovery cache replies with a Discovery_Cache_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpDiscoveryCacheRsp(detailed in Section 9.2.3.14).

Parent topic:ZPS_eAplZdpDiscoveryCacheRequest

Parameters

- ***hAPduInst:*** Handle of APDU instance in which request is sent.
- ****pu8SeqNumber:*** Pointer to sequence number of request.
- ****psZdpDiscoveryCacheReq:*** Pointer to request (see above).

Parent topic:ZPS_eAplZdpDiscoveryCacheRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpDiscoveryCacheRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpDiscoveryStoreRequest

```
ZPS\_teStatus ZPS\_eAplZdpDiscoveryStoreRequest\(
    PDUM\_thAPduInstance hAPduInst,
    ZPS\_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS\_tsAplZdpDiscoveryStoreReq
                *psZdpDiscoveryStoreReq);
```

Description This function can be called on an End Device to request a remote node to reserve memory space to store the local node's 'discovery information'. To do this, the remote node must contain a primary discovery cache. The 'discovery information' includes the local node's IEEE address, network address, Node descriptor, Power descriptor, Simple descriptor and number of active endpoints. The function sends a Discovery_store_req request to the remote node.

This request includes the network and IEEE addresses of the sending node as well as the amount of storage space (in bytes) needed to store the information. The request is represented by the structure below (further detailed in Section 9.2.2.16).

```
typedef struct {
        uint16 u16NwkAddr;
        uint64 u64IeeeAddr;
        uint8 u8NodeDescSize;
        uint8 u8PowerDescSize;
        uint8 u8ActiveEpSize;
        uint8 u8SimpleDescCount;
        /* Rest of message is variable length */
        uint8* pu8SimpleDescSizeList;
} ZPS_tsAplZdpDiscoveryStoreReq;
```

On receiving this request, the remote node first checks whether it has a primary discovery cache. If this is the case, it checks whether it has storage space in the cache for the new discovery information. If the space is available, it is reserved until the information is later uploaded from the local node.

The node replies with a Discovery_store_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpDiscoveryStoreRsp(detailed in Section 9.2.3.15).

Parent topic:ZPS_eAplZdpDiscoveryStoreRequest

Parameters

- ***hAPduInst:******* Handle of APDU instance in which request is sent.
- ***uDstAddr:*** Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr*:** Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- ***pu8SeqNumber*:** Pointer to sequence number of request
- ***psZdpDiscoveryStoreReq:*** Pointer to request (see above)

Parent topic:ZPS_eAplZdpDiscoveryStoreRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpDiscoveryStoreRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpNodeDescStoreRequest

```
ZPS\__teStatus ZPS\__eAplZdpNodeDescStoreRequest\(
    PDUM\__thAPduInstance hAPduInst,
    ZPS\__tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS\__tsAplZdpNodeDescStoreReq
    *psZdpNodeDescStoreReq);
```

Description This function can be called on an End Device to upload the local node's Node descriptor for storage in the primary discovery cache on a remote node. The function sends a Node_Desc_store_req command to the remote node.

This request includes the network and IEEE addresses of the sending node as well as the Node descriptor to store. The request is represented by the structure below (further detailed in Section 9.2.2.17).

typedef struct { uint16 u16NwkAddr; uint64 u64IeeeAddr;

/* Rest of message is variable length */ ZPS_tsAplZdpNodeDescriptor sNodeDescriptor;

} ZPS_tsAplZdpNodeDescStoreReq;

On receiving the request, the remote node will first check whether it has a primary discovery cache. If this is the case, it will check whether it has previously reserved storage space in its cache for the local node. If it has, it will store the Node descriptor in its cache.

The node replies with a Node_Desc_store_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpNodeDescStoreRsp(detailed in Section 9.2.3.16).

Note: This function should only be called if storage space for the local node's 'discovery information' has previously been reserved on the remote node following a call to **ZPS_eAplZdpDiscoveryStoreRequest()**.

Parent topic:ZPS_eAplZdpNodeDescStoreRequest

Parameters

- **hAPduInst:** Handle of APDU instance in which request is sent.
- ***uDstAddr:*** Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr:*** Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- ****pu8SeqNumber*:** Pointer to sequence number of request
- ****psZdpNodeDescStoreReq:*** Pointer to request (see above)

Parent topic:ZPS_eAplZdpNodeDescStoreRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpNodeDescStoreRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpPowerDescStoreRequest

```
ZPS\__teStatus ZPS\__eAplZdpPowerDescStoreRequest\(
    PDUM\__thAPduInstance hAPduInst,
    ZPS\__tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS\__tsAplZdpPowerDescStoreReq
    *psZdpPowerDescStoreReq);
```

Description This function can be called on an End Device to upload the local node's Power descriptor for storage in the primary discovery cache on a remote node. The function sends a Power_Desc_store_req request to the remote node.

This request includes the network and IEEE addresses of the sending node as well as the Power descriptor to store. The request is represented by the structure below (further detailed in Section 9.2.2.18).

```
typedef struct {
    uint16 u16NwkAddr;
    uint64 u64IeeeAddr;
    /* Rest of message is variable length */
    ZPS_tsAplZdpNodePowerDescriptor sPowerDescriptor;
} ZPS_tsAplZdpPowerDescStoreReq;
```

On receiving the request, the remote node first checks whether it has a primary discovery cache. If this is the case, it checks whether it has previously reserved storage space in its cache for the local node. If it has, it stores the Power descriptor in its cache.

The node replies with a Power_Desc_store_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpPowerDescStoreRsp(detailed in Section 9.2.3.17).

Note: This function should only be called if storage space for the local node's 'discovery information' has previously been reserved on the remote node following a call to **ZPS_eAplZdpDiscoveryStoreRequest()**.

Parent topic:ZPS_eAplZdpPowerDescStoreRequest

Parameters

- ***hAPduInst:*** **** Handle of APDU instance in which request is sent.
- ***uDstAddr:*** Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr***: Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- ****pu8SeqNumber:*** Pointer to sequence number of request
- ****psZdpPowerDescStoreReq***: Pointer to request (see above)

Parent topic:ZPS_eAplZdpPowerDescStoreRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3

- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpPowerDescStoreRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpSimpleDescStoreRequest

```
ZPS\_teStatus ZPS\_eAplZdpSimpleDescStoreRequest\(
    PDUM\_thAPduInstance hAPduInst,
    ZPS\_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS\_tsAplZdpSimpleDescStoreReq
                *psZdpSimpleDescStoreReq);
```

Description This function can be called on an End Device to upload a Simple descriptor from the local node for storage in the primary discovery cache on the specified remote node. The Simple descriptor for each endpoint on the local node must be uploaded separately using this function. The function sends a *Simple_Desc_store_req* request to the remote node.

This request includes the network and IEEE addresses of the sending node as well as the Simple descriptor to store. The request is represented by the structure below (further detailed in Section 9.2.2.19).

```
typedef struct {
    uint16 u16NwkAddr;
    uint64 u64IeeeAddr;
    uint8 u8Length;
    /* Rest of message is variable length */
    ZPS_tsAplZdpSimpleDescType sSimpleDescriptor;
} ZPS_tsAplZdpSimpleDescStoreReq;
```

On receiving the request, the remote node first checks whether it has a primary discovery cache. If this is the case, it checks whether it has previously reserved storage space in its cache for the local node. If it has, it stores the Simple descriptor in its cache.

The node replies with a *Simple_Desc_store_rsp* response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpSimpleDescStoreRsp(detailed in Section 9.2.3.18).

Note: This function should only be called if storage space for the local node's 'discovery information' has previously been reserved on the remote node following a call to **ZPS_eAplZdpDiscoveryStoreRequest()**.

Parent topic:ZPS_eAplZdpSimpleDescStoreRequest

Parameters

- ***hAPduInst*:** Handle of APDU instance in which request is sent.
- ***uDstAddr*:** Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr:*** Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- ****pu8SeqNumber:*** Pointer to sequence number of request
- ****psZdpSimpleDescStoreReq*:** Pointer to request (see above)

Parent topic:ZPS_eAplZdpSimpleDescStoreRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpSimpleDescStoreRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpActiveEpStoreRequest

```
ZPS\_teStatus ZPS\_eAplZdpActiveEpStoreRequest\(
    PDUM\_thAPduInstance hAPduInst,
    ZPS\_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS\_tsAplZdpActiveEpStoreReq
            *psZdpActiveEpStoreReq);
```

Description This function can be called on an End Device to upload a list of its active endpoints for storage in the primary discovery cache on a remote node. The function sends an *Active_EP_store_req* command to the remote node.

This request includes the network and IEEE addresses of the sending node as well as the list of active endpoints to store. The request is represented by the structure below (further detailed in Section 9.2.2.20).

```
typedef struct {
    uint16 u16NwkAddr;
    uint64 u64IeeeAddr;
    uint8 u8ActiveEPCount;
    /* Rest of message is variable length */
    uint8* pu8ActiveEpList;
} ZPS_tsAplZdpActiveEpStoreReq;
```

On receiving the request, the remote node first checks whether it has a primary discovery cache. If this is the case, it checks whether it has previously reserved storage space in its cache for the local node. If it has, it stores the list of active endpoints in its cache.

The node replies with an *Active_EP_store_rsp* response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpActiveEpStoreRsp(detailed in Section 9.2.3.19).

Note: This function should only be called if storage space for the local node's 'discovery information' has previously been reserved on the remote node following a call to **ZPS_eAplZdpDiscoveryStoreRequest()**.

Parent topic:ZPS_eAplZdpActiveEpStoreRequest

Parameters

- ***hAPduInst:*** Handle of APDU instance in which request is sent.
- ***uDstAddr:*** Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)

- ***bExtAddr:*** Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- ****pu8SeqNumber:*** Pointer to sequence number of request
- ****psZdpActiveEpStoreReq:*** Pointer to request (see above).

Parent topic:ZPS_eAplZdpActiveEpStoreRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpActiveEpStoreRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpFindNodeCacheRequest

```
ZPS\_teStatus ZPS\_eAplZdpFindNodeCacheRequest\(
  PDUM\_thAPduInstance hAPduInst,
  uint8 *pu8SeqNumber,
  ZPS\_tsAplZdpFindNodeCacheReq
          *psZdpFindNodeCacheReq);
```

Description This function can be used to search for nodes in the network that hold 'discovery information' about a particular node. The function broadcasts a *Find_node_cache_req*request to the network.

This request includes the network and IEEE addresses of the node of interest. The request is represented by the structure below (further detailed in Section 8.2.2.21).

```
typedef struct {
    uint16 u16NwkAddr;
    uint64 u64IeeeAddr;
} ZPS_tsAplZdpFindNodeCacheReq;
```

On receiving the request, a remote node first checks whether it has a primary discovery cache, or is the specified node itself. If either is the case, it checks whether it holds the required information and, if this is the case, replies with a *Find_node_cache_rsp*response. This response should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpFindNodeCacheRsp(detailed in Section 8.2.3.20).

Only nodes that hold the required information respond to the request.

Parent topic:ZPS_eAplZdpFindNodeCacheRequest

Parameters

- ***hAPduInst:*** Handle of APDU instance in which request is sent.
- ****pu8SeqNumber:*** Pointer to sequence number of request
- ****psZdpFindNodeCacheReq:*** Pointer to request (see above)

Parent topic:ZPS_eAplZdpFindNodeCacheRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpFindNodeCacheRequest

Parent topic: Service Discovery functions

ZPS_eAplZdpRemoveNodeCacheRequest

```
ZPS\_teStatus ZPS\_eAplZdpRemoveNodeCacheRequest\(
    PDUM\_thAPduInstance hAPduInst,
    ZPS\_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS\_tsAplZdpRemoveNodeCacheReq
                    *psZdpRemoveNodeCacheReq);
```

Description This function requests a Primary Discovery Cache node to remove from its cache all 'discovery information' relating to a particular End Device. The function sends a *Remove_node_cache_req* request to the Primary Discovery Cache node.

The effect of a successful request is to remove the relevant 'discovery information' and free the corresponding storage space in the cache previously reserved by **ZPS_eAplZdpDiscoveryStoreRequest()**\ (which may have been called from another node in the network).

This request includes the network and IEEE addresses of the End Device whose 'discovery information' is to be removed. The request is represented by the structure below (further detailed in Section 9.2.2.22).

```
typedef struct {
    uint16 u16NwkAddr;
    uint64 u64IeeeAddr;
} ZPS_tsAplZdpRemoveNodeCacheReq;
```

On receiving the request, the remote node first checks whether it has a primary discovery cache. If this is the case, it checks whether it has previously received and implemented a *Discovery_store_req*request for the specified End Device, resulting from a call to **ZPS_eAplZdpDiscoveryStoreRequest()**. If it has, it deletes the relevant data and unreserve the corresponding part of the cache.

The node replies with a *Remove_node_cache_rsp* response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpRemoveNodeCacheRsp(detailed in Section 9.2.3.21).

Parent topic:ZPS_eAplZdpRemoveNodeCacheRequest

Parameters

- ***hAPduInst:*** Handle of APDU instance in which request is sent.
- ***uDstAddr:*** Address of destination node of request(can be 16- or 64-bit, as specified by *bExtAddr*)
- ***bExtAddr:*** Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address

  - FALSE: 16-bit network address
- ***pu8SeqNumber:*** Pointer to sequence number of request
- ***psZdpRemoveNodeCacheReq:*** Pointer to request (see above)

Parent topic:ZPS_eAplZdpRemoveNodeCacheRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpRemoveNodeCacheRequest

Parent topic: Service Discovery functions

Parent topic:ZDP API functions

Binding functions The ZDP Binding functions are concerned with binding nodes together, to aid communication between them, and managing binding tables.

1. ZPS_eAplZdpEndDeviceBindRequest
2. ZPS_eAplZdpBindUnbindRequest
3. ZPS_eAplZdpBindRegisterRequest
4. ZPS_eAplZdpReplaceDeviceRequest
5. ZPS_eAplZdpStoreBkupBindEntryRequest
6. ZPS_eAplZdpRemoveBkupBindEntryRequest
7. ZPS_eAplZdpBackupBindTableRequest
8. ZPS_eAplZdpRecoverBindTableRequest
9. ZPS_eAplZdpBackupSourceBindRequest
10. ZPS_eAplZdpRecoverSourceBindRequest
11. ZPS_eAplAibRemoveBindTableEntryForMacAddress

Note:

1. Some of the above binding functions cannot be used to send requests to nodes that run the NXP ZigBee PRO stack. They are supplied in the NXP ZDP API in order to facilitate interoperability with nodes based on non-NXP software, which supports the corresponding requests. If applicable, this restriction is noted in the function description.
2. Further binding functions are provided in the ZDO API and are described in Section 7.1.1, “Network Deployment Functions”.

ZPS_eAplZdpEndDeviceBindRequest

```
ZPS\__teStatus ZPS\__eAplZdpEndDeviceBindRequest\(
        PDUM\__thAPduInstance hAPduInst,
        uint8 *pu8SeqNumber,
        ZPS\__tsAplZdpEndDeviceBindReq
                *psZdpEndDeviceBindReq);
```

Description

This function sends a binding request to the Coordinator in order to bind an endpoint on the local node to an endpoint on a remote node (these nodes can be End Devices or Routers). The function should normally be invoked as the result of a user action on the local node, such as pressing a button. The function sends an End_Device_Bind_req request to the Coordinator.

This request includes details of the source node, endpoint and clusters. The request is represented by the structure below (further detailed in Section 9.2.2.23).

```
typedef struct {
  uint16 u16BindingTarget;
  uint64 u64SrcIeeeAddress;
  uint8 u8SrcEndpoint;
  uint16 u16ProfileId;
  /* Rest of message is variable length */
  uint8 u8NumInClusters;
  uint16 *pu16InClusterList;
  uint8 u8NumOutClusters;
  uint16 *pu16OutClusterList;
} ZPS_tsAplZdpEndDeviceBindReq;
```

On receiving the request, the Coordinator waits (for a pre-defined timeout period) for another binding request, from a different node, so that it can pair the requests and bind the endpoints. In order to bind the endpoints, their application profile IDs must match, and they must have compatible clusters in their input and output cluster lists.

The Coordinator replies to a binding request with an End_Device_Bind_rsp response, which should be collected on the requesting node using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpEndDeviceBindRsp(detailed in Section 9.2.3.22).

The stack will automatically update the Binding tables on the two End Devices (following further bind requests from the Coordinator) and an ZPS_EVENT_ZDO_BIND event will be generated on the End Devices to signal these updates.

Description This function sends a binding request to the Coordinator in order to bind an endpoint on the local node to an endpoint on a remote node (these nodes can be End Devices or Routers). The function should normally be invoked as the result of a user action on the local node, such as pressing a button. The function sends an *End_Device_Bind_req* request to the Coordinator.

This request includes details of the source node, endpoint and clusters. The request is represented by the structure below (further detailed in Section 9.2.2.23).

```
typedef struct {
  uint16 u16BindingTarget;
  uint64 u64SrcIeeeAddress;
  uint8 u8SrcEndpoint;
  uint16 u16ProfileId;
  /* Rest of message is variable length */
  uint8 u8NumInClusters;
  uint16 *pu16InClusterList;
  uint8 u8NumOutClusters;
  uint16 *pu16OutClusterList;
} ZPS_tsAplZdpEndDeviceBindReq;
```

On receiving the request, the Coordinator waits (for a pre-defined timeout period) for another binding request, from a different node, so that it can pair the requests and bind the endpoints. In order to bind the endpoints, their application profile IDs must match, and they must have compatible clusters in their input and output cluster lists.

The Coordinator replies to a binding request with an *End_Device_Bind_rsp* response, which should be collected on the requesting node using the function **ZQ_bZQueueReceive()** and stored

in a structure of type ZPS_tsAplZdpEndDeviceBindRsp(detailed in Section 9.2.3.22).

The stack automatically updates the Binding tables on the two End Devices (following further bind requests from the Coordinator) and an ZPS_EVENT_ZDO_BIND event is generated on the End Devices to signal these updates.

Parent topic:ZPS_eAplZdpEndDeviceBindRequest

Parameters

- ***hAPduInst:*** Handle of APDU instance in which request is sent.
- ****pu8SeqNumber:*** Pointer to sequence number of request
- ****psZdpEndDeviceBindReq:*** Pointer to request (see above)

Parent topic:ZPS_eAplZdpEndDeviceBindRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpEndDeviceBindRequest

Parent topic: Binding functions

ZPS_eAplZdpBindUnbindRequest

```
ZPS\__teStatus ZPS\__eAplZdpBindUnbindRequest\(
        PDUM\__thAPduInstance hAPduInst,
        ZPS\__tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        bool bBindReq,
        ZPS\__tsAplZdpBindUnbindReq *psZdpBindReq);
```

Description This function sends a binding or unbinding request (as specified) to a remote node which hosts a binding table. The function requests a modification of the binding table in order to bind or unbind two endpoints of nodes in the network. The nodes to be bound/unbound may be different from the node sending the request and the node receiving the request. The latter must be either a node with a primary binding table cache or the source node for the binding. This function could typically be used in a commissioning application to configure bindings between nodes during system set- up.

The function sends a Bind_req or Unbind_req request to the remote node which hosts the binding table to be modified. This request includes details of the source node and endpoint, and the target node and endpoint for the binding. The request is represented by the structure below (further detailed in Section 9.2.2.24).

```
typedef struct {
    uint64 u64SrcAddress;
    uint8 u8SrcEndpoint;
    uint16 u16ClusterId;
    uint8 u8DstAddrMode;
    union {
        struct {
```

(continues on next page)

(continued from previous page)

```
            uint16 u16DstAddress;
        } sShort;
        struct {
            uint64 u64DstAddress;
            uint8 u8DstEndPoint;
        } sExtended;
    } uAddressField;
} ZPS_tsAplZdpBindUnbindReq;
```

On receiving the request, the remote node adds or removes the relevant entry in its binding table and locally generates the event ZPS_EVENT_ZDO_BIND or ZPS_EVENT_ZDO_UNBIND, as appropriate, to signal the relevant update.

If the remote node holds a primary binding table cache, it checks whether the source node for the binding holds a table of its own source bindings (see the description of **ZPS_eAplZdpBindRegisterRequest()**). If it is so, it automatically requests an update of this table. A node with a primary binding table cache also requests an update of the back-up cache, if one exists.

The remote node replies with a Bind_rsp or Unbind_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpBindRsp(detailed in Section 9.2.3.23) or ZPS_tsAplZdpUnbindRsp(detailed in Section 9.2.3.24).

Parent topic:ZPS_eAplZdpBindUnbindRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- *bBindReq* Bind or unbind request:
  - TRUE: bind
  - FALSE: unbind
- * *psZdpBindReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpBindUnbindRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpBindUnbindRequest

Parent topic: Binding functions

ZPS_eAplZdpBindRegisterRequest

```
ZPS_teStatus ZPS_eAplZdpBindRegisterRequest(
    PDUM_thAPduInstance hAPduInst,
    ZPS_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS_tsAplZdpBindRegisterReq *psZdpBindRegisterReq);
```

Description This function informs a remote node with a primary binding table cache that the local node will hold its own binding table entries (and therefore the remote node does not need to hold these entries). The function sends a Bind_Register_req request to the remote node.

The IEEE address of the local node must be specified in the request, which is represented by the structure below (further detailed in Section 8.2.2.25).

typedef struct {

uint64 u64NodeAddress;

} ZPS_tsAplZdpBindRegisterReq;

The remote node replies with a Bind_Register_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpBindRegisterRsp(detailed in Section 9.2.3.25). This response contains any information stored about the binding on the remote.

Parent topic:ZPS_eAplZdpBindRegisterRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpPowerDescReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpBindRegisterRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpBindRegisterRequest

Parent topic: Binding functions

ZPS_eAplZdpReplaceDeviceRequest

```
ZPS_teStatus ZPS_eAplZdpReplaceDeviceRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
  ZPS_tsAplZdpReplaceDeviceReq *psZdpReplaceDeviceReq);
```

Description This function requests a remote node with a primary binding table cache to modify binding table entries with new data - more specifically, binding table entries can be modified by replacing an IEEE address and/or associated endpoint number. This function could typically be used in a commissioning application to modify bindings between nodes. The function sends a Replace_Device_req request to the remote node.

This request must include the old IEEE address and its replacement, as well as the corresponding endpoint number and its replacement (if any). The request is represented by the structure below (further detailed in Section 9.2.2.26).

typedef struct {

uint64 u64OldAddress;

uint8 u8OldEndPoint;

uint64 u64NewAddress;

uint8 u8NewEndPoint;

} ZPS_tsAplZdpReplaceDeviceReq;

On receiving this request, the remote node will search its binding table for entries containing the old IEEE address and old endpoint number from the request - this pair of values may make up the source or destination data of the binding table entry.

These values will be replaced by the new IEEE address and endpoint number from the request. Note that if the endpoint number in the request is zero, only the address will be included in the 'search and replace' (the endpoint number in the modified binding table entries will be left unchanged).

The remote node will check whether a node affected by a binding table change holds a table of its own source bindings (see **ZPS_eAplZdpBindRegisterRequest()**) and, if so, automatically requests an update of this table. The remote node will also request an update of the back-up of the primary binding table cache, if one exists.

The remote node will reply with a Replace_Device_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpReplaceDeviceRsp(detailed in Section 9.2.3.26).

Parent topic:ZPS_eAplZdpReplaceDeviceRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpReplaceDeviceReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpReplaceDeviceRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpReplaceDeviceRequest

Parent topic: Binding functions

ZPS_eAplZdpStoreBkupBindEntryRequest

```
ZPS_teStatus ZPS_eAplZdpStoreBkupBindEntryRequest(
        PDUM_thAPdu hAPdu,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        uint16 u16ProfileId,
        ZPS_tsAplZdpStoreBkupBindEntryReq
                *psZdpStoreBkupBindEntryReq);
```

Description This function requests that a back-up of an entry in the local primary binding table cache is performed on a remote node. The destination node of the request must hold the corresponding back-up binding table cache. The back-up operation is normally required when a new entry has been added to the primary binding table cache.

Note: This function is provided in the NXP ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

This request must include the binding table entry to be backed up. The request is represented by the structure below (further detailed in Section 9.2.2.27).

```
typedef struct {
      uint64 u64SrcAddress;
      uint8 u8SrcEndPoint;
      uint16 u16ClusterId;
      uint8 u8DstAddrMode;
      union {
        struct {
            uint16 u16DstAddress;
              } sShort;
      struct {
      uint64 u64DstAddress;
      uint8 u8DstEndPoint;
      } sExtended;
    };
} ZPS_tsAplZdpStoreBkupBindEntryReq;
```

On receiving the request, the remote node adds the specified binding table entry to its back-up binding table cache, if possible.

The remote node replies with a Store_Bkup_Bind_Entry_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpStoreBkupBindEntryRsp(detailed in Section 9.2.3.27).

Parent topic:ZPS_eAplZdpStoreBkupBindEntryRequest

Parameters

- *hAPdu* Handle of APDU in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- *u16ProfileId* Application profile ID
- * *psZdpStoreBkupBindEntryReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpStoreBkupBindEntryRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpStoreBkupBindEntryRequest

Parent topic: Binding functions

ZPS_eAplZdpRemoveBkupBindEntryRequest

```
ZPS_teStatus ZPS_eAplZdpRemoveBkupBindEntryRequest(
    PDUM_thAPduInstance hAPduInst,
    ZPS_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS_tsAplZdpRemoveBkupBindEntryReq
            *psZdpRemoveBkupBindEntryReq);
```

Description This function requests the removal of an entry in the back-up binding table cache on a remote node. The function must be called from the node with the corresponding primary binding table cache. The removal of a back-up entry is normally required when an entry in the primary binding table cache has been removed.

Note: This function is provided in the NXP ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

This request must include the binding table entry to be removed. The request is represented by the structure below (further detailed in Section 9.2.2.28).

typedef struct {

uint64 u64SrcAddress; uint8 u8SrcEndPoint; uint16 u16ClusterId; uint8 u8DstAddrMode; union {

struct {

uint16 u16DstAddress;

} sShort; struct {

uint64 u64DstAddress; uint8 u8DstEndPoint;

} sExtended;

};

} ZPS_tsAplZdpRemoveBkupBindEntryReq;

On receiving the request, the remote node removes the specified binding table entry from its back-up binding table cache, if possible.

The remote node replies with a Remove_Bkup_Bind_Entry_rsp response, which should be collected using the function **ZQ_bZQueueReceive()**and stored in a structure of type ZPS_tsAplZdpRemoveBkupBindEntryRsp(detailed in Section9.2.3.28).

Parent topic:ZPS_eAplZdpRemoveBkupBindEntryRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpRemoveBkupBindEntryReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpRemoveBkupBindEntryRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpRemoveBkupBindEntryRequest

Parent topic: Binding functions

ZPS_eAplZdpBackupBindTableRequest

```
ZPS_teStatus ZPS_eAplZdpBackupBindTableRequest(
  PDUM_thAPduInstance hAPduInst,
  ZPS_tuAddress uDstAddr,
  bool bExtAddr,
  uint8 *pu8SeqNumber,
  ZPS_tsAplZdpBackupBindTableReq
                    *psZdpBackupBindTableReq);
```

Description This function requests that a back-up of the locally held primary binding table cache is performed on a remote node - the whole or part of the table can be backed up. The destination node of the request must hold the corresponding back-up binding table cache. The latter must already exist and be associated with the cache on the local node through a previous discovery.

Note: This function is provided in the NXP ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

This request must include the binding table entries to be backed up. The request is represented by the structure below (further detailed in Section 9.2.2.29).

typedef struct {

uint16 u16BindingTableEntries; uint16 u16StartIndex;

uint16 u16BindingTableListCount;

/* Rest of message is variable length */ ZPS_tsAplZdpBindingTable sBindingTable;

} ZPS_tsAplZdpBackupBindTableReq;

On receiving the request, the remote node saves the new binding table, if possible, overwriting existing entries. If the new table is longer than the previous one, as many extra entries as possible will be saved.

The remote node replies with a Backup_Bind_Table_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpBackupBindTableRsp(detailed in Section 9.2.3.29).

Parent topic:ZPS_eAplZdpBackupBindTableRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpBackupBindTableReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpBackupBindTableRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpBackupBindTableRequest

Parent topic: Binding functions

ZPS_eAplZdpRecoverBindTableRequest

```
ZPS_teStatus ZPS_eAplZdpRecoverBindTableRequest(
    PDUM_thAPduInstance hAPduInst,
    ZPS_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS_tsAplZdpRecoverBindTableReq
                *psZdpRecoverBindTableReq);
```

Description This function requests that a back-up of the locally held primary binding table cache is recovered from a remote node. The destination node of the request must hold the back-up binding table cache which is associated with the primary cache on the local node.

Note: This function is provided in the NXP ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

This request must indicate the starting index in the binding table for the recovery. The request is represented by the structure below (further detailed in Section 9.2.2.30).

typedef struct {

uint16 u16StartIndex;

} ZPS_tsAplZdpRecoverBindTableReq;

The remote node replies with a Recover_Bind_Table_rsp response containing the required binding table entries, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpRecoverBindTableRsp(detailed in Section 9.2.3.30). As many binding entries as possible are included in this response. If the returned binding table is incomplete, this is indicated in the response and this function must be called again, with the appropriate starting index, to recover the rest of the table.

Parent topic:ZPS_eAplZdpRecoverBindTableRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpRecoverBindTableReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpRecoverBindTableRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpRecoverBindTableRequest

Parent topic: Binding functions

ZPS_eAplZdpBackupSourceBindRequest

```
ZPS_teStatus ZPS_eAplZdpBackupSourceBindRequest(
 PDUM_thAPduInstance hAPduInst,
 ZPS_tuAddress uDstAddr,
 bool bExtAddr,
 uint8 *pu8SeqNumber,
 ZPS_tsAplZdpBackupSourceBindReq
              *psZdpBackupSourceBindReq);
```

Description This function requests that a back-up of the locally held source binding table is performed on a remote node. This source binding table contains entries only relevant to the local node. The function must be called from a node with a primary binding table cache and the destination node of the request must hold the corresponding back-up binding table cache.

Note: This function is provided in the NXP ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

This request must include the source binding table entries to be backed up. The request is represented by the structure below (further detailed in Section 9.2.2.31).

typedef struct {

uint16 u16SourceTableEntries; uint16 u16StartIndex;

uint16 u16SourceTableListCount;

/* Rest of message is variable length */ uint64* pu64SourceAddress;

} ZPS_tsAplZdpBackupSourceBindReq;

On receiving the request, the remote node saves the new source binding table, if possible, overwriting existing entries. If the new table is longer than the previous one, as many extra entries as possible will be saved.

The remote node replies with a Backup_Source_Bind_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpBackupSourceBindRsp(detailed in Section 9.2.3.31).

Parent topic:ZPS_eAplZdpBackupSourceBindRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpBackupSourceBindReq*Pointer to request (see above)

Parent topic:ZPS_eAplZdpBackupSourceBindRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpBackupSourceBindRequest

Parent topic: Binding functions

ZPS_eAplZdpRecoverSourceBindRequest

```
ZPS_teStatus ZPS_eAplZdpBackupSourceBindRequest(
    PDUM_thAPduInstance hAPduInst,
    ZPS_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS_tsAplZdpBackupSourceBindReq
                *psZdpBackupSourceBindReq);
```

Description This function requests that a back-up of the locally held source binding table is recovered from a remote node. The function must be called from a node with a primary binding table cache and the destination node of the request must hold the corresponding back-up binding table cache.

Note: This function is provided in the NXP ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

This request must indicate the starting index in the binding table for the recovery. The request is represented by the structure below (further detailed in Section 9.2.2.32).

typedef struct {

uint16 u16StartIndex;

} ZPS_tsAplZdpRecoverSourceBindReq;

The remote node replies with a Recover_Source_Bind_rsp response containing the required binding table entries, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpRecoverSourceBindRsp(detailed in Section 9.2.3.32). As many binding entries as possible are included in this response. If the returned binding table is incomplete, this is indicated in the response and this function must be called again, with the appropriate starting index, to recover the rest of the table.

Parent topic:ZPS_eAplZdpRecoverSourceBindRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request

- * *psZdpRecoverSourceBindReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpRecoverSourceBindRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpRecoverSourceBindRequest

Parent topic: Binding functions

ZPS_eAplAibRemoveBindTableEntryForMacAddress

```
ZPS_teStatus
    ZPS_eAplAibRemoveBindTableEntryForMacAddress(
                        uint64 u64MacAddress);
```

Description This function requests the removal of the entry corresponding to the specified IEEE/ MAC address from the local binding table.

Parent topic:ZPS_eAplAibRemoveBindTableEntryForMacAddress

Parameters *u64MacAddress* IEEE/MAC address contained in the binding table entry to be removed

Parent topic:ZPS_eAplAibRemoveBindTableEntryForMacAddress

Returns ZPS_E_SUCCESS

Parent topic:ZPS_eAplAibRemoveBindTableEntryForMacAddress

Parent topic: Binding functions

Parent topic:ZDP API functions

Network Management Services functions The ZDP Network Management Services functions are concerned with requests for network operations to be implemented remotely.

The functions are listed below.

1. ZPS_eAplZdpMgmtNwkDiscRequest
2. ZPS_eAplZdpMgmtLqiRequest
3. ZPS_eAplZdpMgmtRtgRequest
4. ZPS_eAplZdpMgmtBindRequest
5. ZPS_eAplZdpMgmtLeaveRequest
6. ZPS_eAplZdpMgmtDirectJoinRequest
7. ZPS_eAplZdpMgmtPermitJoiningRequest
8. ZPS_eAplZdpMgmtCacheRequest
9. ZPS_eAplZdpMgmtNwkUpdateRequest

10. ZPS_eAplZdpParentAnnceReq

Note: Some of these functions cannot be used to send requests to nodes that run the NXP ZigBee PRO stack. They are supplied in the ZDP API in order to facilitate interoperability with nodes based on non-NXP software which supports the corresponding requests.

ZPS_eAplZdpMgmtNwkDiscRequest

```
ZPS_teStatus ZPS_eAplZdpMgmtNwkDiscRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpMgmtNwkDiscReq
                *psZdpMgmtNwkDiscReq);
```

Description This function requests a remote node to perform a channel scan in order to discover any other wireless networks that are operating in the neighborhood.

Note: This function is provided in the ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status **ZPS_ZDP_NOT_SUPPORTED**.

This request must specify the requirements for the scan: channels to scan, duration of scan, starting channel. The request is represented by the structure below (further detailed in Section 8.2.2.33).

typedef struct {

uint32 u32ScanChannels;

uint8 u8ScanDuration;

uint8 u8StartIndex;

} ZPS_tsAplZdpMgmtNwkDiscReq;

The remote node replies with a Mgmt_NWK_Disc_rsp response containing the scan results, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpMgmtNwkDiscRsp(detailed in Section 8.2.3.33).

Parent topic:ZPS_eAplZdpMgmtNwkDiscRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpMgmtNwkDiscReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpMgmtNwkDiscRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpMgmtNwkDiscRequest

Parent topic: Network Management Services functions

ZPS_eAplZdpMgmtLqiRequest

```
ZPS_teStatus ZPS_eAplZdpMgmtLqiRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpMgmtLqiReq *psZdpMgmtLqiReq);
```

Description This function requests a remote node to provide a list of neighboring nodes, from its Neighbor table, including LQI (link quality) values for radio transmissions from each of these nodes. The destination node of this request must be a Router or the Co- ordinator.

This request must specify the index of the first node in the Neighbor table to report. The request is represented by the structure below (further detailed in Section 8.2.2.34).

typedef struct {

uint8 u8StartIndex;

} ZPS_tsAplZdpMgmtLqiReq;

The remote node replies with a Mgmt_Lqi_rsp response containing the required information, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpMgmtLqiRsp(detailed in Section 8.2.3.34).

Parent topic:ZPS_eAplZdpMgmtLqiRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpMgmtLqiReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpMgmtLqiRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpMgmtLqiRequest

Parent topic: Network Management Services functions

ZPS_eAplZdpMgmtRtgRequest

```
ZPS_teStatus ZPS_eAplZdpMgmtRtgRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpMgmtRtgReq *psZdpMgmtRtgReq);
```

Description This function requests a remote node to provide the contents of its Routing table. The destination node of this request must be a Router or the Coordinator.

This request must specify the index of the first entry in the Routing table to report. The request is represented by the structure below (further detailed in Section 8.2.2.35).

typedef struct {

uint8 u8StartIndex;

} ZPS_tsAplZdpMgmtRtgReq;

The remote node replies with a Mgmt_Rtg_rsp response containing the required information, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpMgmtRtgRsp(detailed in Section 8.2.3.35).

Parent topic:ZPS_eAplZdpMgmtRtgRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpMgmtRtgReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpMgmtRtgRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3

- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpMgmtRtgRequest

Parent topic:ZPS_eAplZdpMgmtBindRequest

Parent topic: Network Management Services functions

ZPS_eAplZdpMgmtBindRequest

```
ZPS_teStatus ZPS_eAplZdpMgmtBindRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpMgmtBindReq *psZdpMgmtBindReq);
```

Description This function requests a remote node to provide the contents of its Binding table. The destination node of this request must be a Router or the Coordinator.

This request must specify the index of the first entry in the Binding table to report. The request is represented by the structure below (further detailed in Section8.2.2.36).

typedef struct {

uint8 u8StartIndex;

} ZPS_tsAplZdpMgmtBindReq;

The remote node replies with a Mgmt_Bind_rsp response containing the required information, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpMgmtBindRsp(detailed in Section8.2.3.36).

Parent topic:ZPS_eAplZdpMgmtBindRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpMgmtBindReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpMgmtBindRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpMgmtRtgRequest

Parent topic:ZPS_eAplZdpMgmtBindRequest

Parent topic: Network Management Services functions

ZPS_eAplZdpMgmtLeaveRequest

```
ZPS_teStatus ZPS_eAplZdpMgmtLeaveRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpMgmtLeaveReq *psZdpMgmtLeaveReq);
```

Description This function requests a remote node to leave the network. The request also indicates whether the children of the leaving node should also be requested to leave and whether the leaving node(s) should subsequently attempt to rejoin the network.

Note: This function is provided in the ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status **ZPS_ZDP_NOT_SUPPORTED**.

The IEEE address of the node to leave the network must be included in the request, as well as flags indicating the children and rejoin choices (see above). The request is represented by the structure below (further detailed in Section 8.2.2.37).

typedef struct {

uint64 u64DeviceAddress;

uint8 u8Flags;

} ZPS_tsAplZdpMgmtLeaveReq;

The remote node replies with a Mgmt_Leave_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpMgmtLeaveRsp(detailed in Section 8.2.3.37).

Parent topic:ZPS_eAplZdpMgmtLeaveRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpMgmtLeaveReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpMgmtLeaveRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpMgmtLeaveRequest

Parent topic:ZPS_eAplZdpMgmtDirectJoinRequest

Parent topic:ZPS_eAplZdpMgmtPermitJoiningRequest

Parent topic:ZPS_eAplZdpMgmtCacheRequest

Parent topic: Network Management Services functions

ZPS_eAplZdpMgmtDirectJoinRequest

```
ZPS_teStatus ZPS_eAplZdpMgmtDirectJoinRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpMgmtDirectJoinReq
        *psZdpMgmtDirectJoinReq);
```

Description This function requests a remote node to allow a particular device (identified through its IEEE address) to join the network as a child of the node. Thus, joining should be enabled on the remote node just for the nominated device. The destination node of this request must be a Router or the Coordinator.

Note: This function is provided in the ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status **ZPS_ZDP_NOT_SUPPORTED**.

The IEEE address of the nominated device as well as its capabilities must be included in the request. The request is represented by the structure below (further detailed in Section 8.2.2.38).

typedef struct {

uint64 u64DeviceAddress; uint8 u8Capability;

} ZPS_tsAplZdpMgmtDirectJoinReq;

The remote node replies with a Mgmt_Direct_Join_req response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpMgmtDirectJoinRsp(detailed in Section 8.2.3.38).

Parent topic:ZPS_eAplZdpMgmtDirectJoinRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address

- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpMgmtDirectJoinReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpMgmtDirectJoinRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpMgmtLeaveRequest

Parent topic:ZPS_eAplZdpMgmtDirectJoinRequest

Parent topic:ZPS_eAplZdpMgmtPermitJoiningRequest

Parent topic:ZPS_eAplZdpMgmtCacheRequest

Parent topic: Network Management Services functions

ZPS_eAplZdpMgmtPermitJoiningRequest

```
ZPS_teStatus ZPS_eAplZdpMgmtPermitJoiningRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpMgmtPermitJoiningReq
        *psZdpMgmtPermitJoiningReq);
```

Description This function requests a remote node to enable or disable joining for a specified amount of time. The destination node of this request must be a Router or the Co- ordinator. The request can be unicast to a particular node or broadcast to all routing nodes (for which the destination address must be set to the 16-bit network address 0xFFFC).

Note: This function is provided in the ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status **ZPS_ZDP_NOT_SUPPORTED**.

The duration of the enable or disable joining state must be specified in the request. The request is represented by the structure below (further detailed in Section8.2.2.39).

typedef struct {

uint8 u8PermitDuration; bool_t bTcSignificance;

} ZPS_tsAplZdpMgmtPermitJoiningReq;

If the request was unicast, the remote node replies with a Mgmt_Permit_Joining_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpMgmtPermitJoiningRsp(detailed in Section 8.2.3.39).

Parent topic:ZPS_eAplZdpMgmtPermitJoiningRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.

- *uDstAddr* Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpMgmtPermitJoiningReq*Pointer to request (see above)

Parent topic:ZPS_eAplZdpMgmtPermitJoiningRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpMgmtLeaveRequest

Parent topic:ZPS_eAplZdpMgmtDirectJoinRequest

Parent topic:ZPS_eAplZdpMgmtPermitJoiningRequest

Parent topic:ZPS_eAplZdpMgmtCacheRequest

Parent topic: Network Management Services functions

ZPS_eAplZdpMgmtCacheRequest

```
ZPS_teStatus ZPS_eAplZdpMgmtCacheRequest(
    PDUM_thAPduInstance hAPduInst,
    ZPS_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS_tsAplZdpMgmtCacheReq *psZdpMgmtCacheReq);
```

Description This function requests a remote node to provide a list of the End Devices registered in its primary discovery cache. Therefore, the destination node must contain a primary discovery cache.

Note: This function is provided in the ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status **ZPS_ZDP_NOT_SUPPORTED**.

The request is represented by the structure below (further detailed in Section9.2.2.40).

typedef struct {

uint8 u8StartIndex;

} ZPS_tsAplZdpMgmtCacheReq;

The remote node replies with a Mgmt_Cache_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpMgmtCacheRsp(detailed in Section 9.2.3.40).

Parent topic:ZPS_eAplZdpMgmtCacheRequest

Parameters

- *hAPduInst* Handle of APDU in which request is sent.
- *uDstAddr* Address of destination node of request
- (can be 16- or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpMgmtCacheReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpMgmtCacheRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpMgmtLeaveRequest

Parent topic:ZPS_eAplZdpMgmtDirectJoinRequest

Parent topic:ZPS_eAplZdpMgmtPermitJoiningRequest

Parent topic:ZPS_eAplZdpMgmtCacheRequest

Parent topic: Network Management Services functions

ZPS_eAplZdpMgmtNwkUpdateRequest

```
ZPS_teStatus ZPS_eAplZdpMgmtNwkUpdateRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpMgmtNwkUpdateReq
                    *psZdpMgmtNwkUpdateReq);
```

Description This function requests an update of network parameters related to radio communication. The request can specify any of the following:

- update the radio channel mask (for scans) and the 16-bit network address of the network manager (node nominated to manage radio-band operation of network)
- change the radio channel used
- scan radio channels and report the results

The request can be broadcast or unicast to nodes with radio receivers that are configured to remain on during idle periods.

The request is represented by the structure below (further detailed in Section9.2.2.41).

typedef struct {

uint32 u32ScanChannels;

uint8 u8ScanDuration;

uint8 u8ScanCount;

uint8 u8NwkUpdateId;

uint16 u16NwkManagerAddr;

} ZPS_tsAplZdpMgmtNwkUpdateReq;

The specific action to be taken as a result of this request is indicated through the element u8ScanDuration, as described in the table below.

**u8Sca	**Action**
0x00-0x05	Perform radio channel scan on the set of channels specified through u32ScanChannels. The time, in seconds, spent scanning each channel is determined by the value of u8ScanDurationand the number of scans is equal to the value of u8ScanCount. Valid for unicasts only.
0x06-0xFD	Reserved
0xFE	Change radio channel to single channel specified through u32ScanChannelsand set the network man- ager address to that specified through u16NwkManag- erAddr. Valid for broadcasts only.
0xFF	Update the stored radio channel mask with that specified through u32ScanChannels(but do not scan).Valid for broadcasts only.

The remote node replies with a Mgmt_NWK_Update_notify notification, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type ZPS_tsAplZdpMgmtNwkUpdateNotify(detailed in Section 9.2.3.41).

Parent topic:ZPS_eAplZdpMgmtNwkUpdateRequest

Parameters

- *hAPduInst* Handle of APDU instance in which request is sent.
- *uDstAddr* Address of destination node of request (can be 16-bit or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address:
  - TRUE: 64-bit IEEE (MAC) address
  - FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of request
- * *psZdpMgmtNwkUpdateReq* Pointer to request (see above)

Parent topic:ZPS_eAplZdpMgmtNwkUpdateRequest

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpMgmtNwkUpdateRequest

Parent topic: Network Management Services functions

ZPS_eAplZdpParentAnnceReq

```
ZPS_teStatus ZPS_eAplZdpParentAnnceReq(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpParentAnnceReq *psZdpParentAnnceReq);
```

Description This function is used on a Router or the Coordinator to send a Parent Announcement message to one or more other nodes. In this announcement, the originating node declares which nodes it has as children. These child nodes are specified using their IEEE/MAC addresses.

The message contains the above data in following structure (further detailed in Section 8.2.2.42):

typedef struct {

uint8 u8NumberOfChildren;

uint64* pu64ChildList;

} ZPS_tsAplZdpParentAnnceReq;

If a node which receives this message also has one of the specified nodes as its child (so there is a conflict), the receiving node broadcasts a response to indicate this. The response data is contained in the structure below (further detailed in Section 8.2.3.42):

typedef struct {

uint64* pu64ChildList; uint8 u8NumberOfChildren; uint8 u8Status;

} ZPS_tsAplZdpParentAnnceRsp;

Parent topic:ZPS_eAplZdpParentAnnceReq

Parameters

- *hAPduInst* Handle of APDU instance in which message is sent.
- *uDstAddr* Address of destination node of message (16-bit or 64-bit, as specified by *bExtAddr*)
- *bExtAddr* Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
- * *pu8SeqNumber* Pointer to sequence number of message
- * *psZdpParentAnnceReq* Pointer to message (see above)

Parent topic:ZPS_eAplZdpParentAnnceReq

Returns

- ZPS_E_SUCCESS (request successfully sent)
- APS return codes, listed and described in Section 11.2.2
- NWK return codes, listed and described in Section 11.2.3
- MAC return codes, listed and described in Section 11.2.4

Parent topic:ZPS_eAplZdpParentAnnceReq

Parent topic: Network Management Services functions

Parent topic:ZDP API functions

Response Data Extraction Function The ZDP Response Data Extraction function is concerned with obtaining the data from a received response packet which is destined for the ZDO. The function should be called when a ZPS_EVENT_APS_DATA_INDICATION event is generated for destination endpoint 0.

Note: This function and the related structure ZPS_tsAfZdpEvent are defined in the header file

appZdpExtraction.h.

The function is listed below:

ZPS_bAplZdpUnpackResponse ZPS_bAplZdpUnpackResponse

Function Page **Parent topic:**Response Data Extraction Function

ZPS_bAplZdpUnpackResponse

```
bool ZPS_bAplZdpUnpackResponse(
        ZPS_tsAfEvent *psZdoServerEvent,
        ZPS_tsAfZdpEvent *psReturnStruct);
```

Description This function can be used to extract data received in a response packet which is destined for the ZDO (at endpoint 0). When such a packet is received, the event ZPS_EVENT_APS_DATA_INDICATION is generated. The application must then check whether the destination endpoint number is 0 in the event and, if this is the case, call this function to extract the response data from the event.

A pointer to a ZPS_tsAfZdpEventstructure must be provided, which the function will populate with the extracted data.

Parent topic:ZPS_bAplZdpUnpackResponse

Parameters

- * *psZdoServerEvent* Pointer to structure containing the event (see Section 7.2.2.1)
- * *psReturnStruct* Pointer to structure to receive extracted data (see Section 7.2.2.25)

Parent topic:ZPS_bAplZdpUnpackResponse

Returns

- TRUE if data successfully extracted
- FALSE if data not successfully extracted

Parent topic:ZPS_bAplZdpUnpackResponse

Parent topic: Response Data Extraction Function

Parent topic:ZDP API functions

Parent topic:*ZigBee Device Profile (ZDP) API*

ZDP structures This section describes the structures used by the ZigBee Device Profile (ZDP) API. Three sets of structures are presented:

- Structures used to represent the descriptors that reside on a node - see Section9.2.1
- Structures used to issue requests using the ZDP functions - see Section 9.2.2
- Structures used to receive responses to the ZDP requests - see Section 9.2.3

Descriptor structures These structures are used to represent the following descriptors that contain information about the host node:

- Node descriptor
- Node Power descriptor
- Simple descriptor

The structures are listed below.

- ZPS_tsAplZdpNodeDescriptor
- ZPS_tsAplZdpNodePowerDescriptor
- ZPS_tsAplZdpSimpleDescType

ZPS_tsAplZdpNodeDescriptor The ZDP Node descriptor structure ZPS_tsAplZdpNodeDescriptor is shown below.

```
typedef struct {
    union
    {
        ZPS_tsAplZdpNodeDescBitFields sBitFields;
        uint16 u16Value;
    } uBitUnion;
    uint8 u8MacFlags;
    uint16 u16ManufacturerCode;
    uint8 u8MaxBufferSize;
    uint16 u16MaxRxSize;
    uint16 u16ServerMask;
    uint16 u16MaxTxSize;
    uint8 u8DescriptorCapability;
} ZPS_tsAplZdpNodeDescriptor;
```

where:

- sBitFields is a structure of the type ZPS_tsAplZdpNodeDescBitFields

(described below) containing various items of information about the node.

- u16Value is used for the union and should be set to 0x0000.
- eMacFlags contains 8 bits (bits 0-7) indicating the node capabilities, as required by the IEEE 802.15.4 MAC sub-layer. These node capability flags are described in Table 8.
- u16ManufacturerCode contains 16 bits (bits 0-15) indicating the manufacturer code for the node, where this code is allocated to the manufacturer by the ZigBee Alliance.
- u8MaxBufferSize is the maximum size, in bytes, of an NPDU (Network Protocol Data Unit).
- u16MaxRxSize is the maximum size, in bytes, of an APDU (Application Protocol Data Unit). This value can be greater than the value of u8MaxBufferSize, due to the fragmentation of an APDU into NPDUs.
- u16ServerMask contains 8 bits (bits 0-7) indicating the server status of the node. This server mask is detailed in Table 18 on page 389.
- u16MaxTxSize is the maximum size, in bytes, of the ASDU (Application Sub- layer Data Unit) in which a message can be sent (the message may actually be transmitted in smaller fragments)
- u8DescriptorCapability contains 8 bits (bits 0-7) indicating the properties of the node that can be used by other nodes in network discovery, as indicated in the table below.

Bit	Description
0	Set to 1 if Extended Active Endpoint List is available on the node, 0 otherwise.
1	Set to 1 if Extended Simple Descriptor List is available on the node, 0 otherwise.
2-7	Reserved

ZPS_tsAplZdpNodeDescBitFields The ZPS_tsAplZdpNodeDescBitFieldsstructure is used by the sBitFields

element in the Node descriptor structure (see above), and is shown below:

```
typedef struct {
    unsigned eFrequencyBand : 5;
    unsigned eApsFlags : 3;
    unsigned eReserved : 3; /* reserved */
    unsigned bUserDescAvail : 1;
    unsigned bComplexDescAvail : 1;
    unsigned eLogicalType : 3;
}ZPS_tsAplZdpNodeDescBitFields;
```

where:

- eFrequencyBandis a 5-bit value representing the IEEE 802.15.4 radio- frequency band used by the node:
  - 0: 868-MHz band
  - 2: 915-MHz band
  - 3: 2400-MHz band
- eApsFlagsis a 3-bit value containing flags that indicate the ZigBee APS capabilities of the node (not currently supported and should be set to 0).
- eReservedis a 3-bit reserved value.
- bUserDescAvailis a 1-bit value indicating whether a User descriptor is available for the node - 1 indicates available, 0 indicates unavailable.
- bComplexDescAvailis a 1-bit value indicating whether a Complex descriptor is available for the node - 1 indicates available, 0 indicates unavailable.
- eLogicalTypeis a 3-bit value indicating the ZigBee device of the node:
  - 0: Coordinator
  - 1: Router
  - 2: End Device

Parent topic:ZPS_tsAplZdpNodeDescriptor

Parent topic: Descriptor structures

ZPS_tsAplZdpNodePowerDescriptor The ZDP Node Power descriptor structure ZPS_tsAplZdpNodePowerDescriptoris shown below.

```
typedef struct {
    union
    {
        ZPS_tsAplZdpPowerDescBitFields sBitFields;
        uint16 u16Value;
    }uBitUnion;
} ZPS_tsAplZdpNodePowerDescriptor;
```

where:

- sBitFieldsis a structure of type ZPS_tsAplZdpPowerDescBitFields(described below) containing various items of information about the node's power.
- u16valueis used for the union and should be set to 0x0000.

ZPS_tsAplZdpPowerDescBitFields The ZPS_tsAplZdpPowerDescBitFieldsstructure is used by the sBitFields

element in the Node Power descriptor structure (see above), and is shown below:

```
typedef struct {
    unsigned eCurrentPowerSourceLevel : 4;
    unsigned eCurrentPowerSource : 4;
    unsigned eAvailablePowerSource : 4;
    unsigned eCurrentPowerMode : 4;
}ZPS_tsAplZdpPowerDescBitFields;
```

where:

- eCurrentPowerSourceLevelis a 4-bit value roughly indicating the level of charge of the node's power source (mainly useful for batteries), as follows:
  - 0000: Critically low
  - 0100: Approximately 33%
  - 1000: Approximately 66%
  - 1100: Approximately 100% (near fully charged)
- eCurrentPowerSourceis a 4-bit value indicating the current power source for the node, as detailed below (the bit corresponding to the current power source is set to 1, all other bits are set to 0):
  - Bit 0: Permanent mains supply
  - Bit 1: Rechargeable battery
  - Bit 2: Disposable battery
  - Bit 4: Reserved
- eAvailablePowerSourceis a 4-bit value indicating the available power sources for the node, as detailed above (a bit is set to 1 if the corresponding power source is available).
- eCurrentPowerModeis a 4-bit value indicating the power mode currently used by the node, as follows:
  - 0000: Receiver synchronized with the "receiver on when idle" subfield of the Node descriptor
  - 0001: Receiver switched on periodically, as defined by the Node Power descriptor
  - 0010: Receiver switched on when stimulated, for example, by pressing a button
  - All other values are reserved.

Parent topic:ZPS_tsAplZdpNodePowerDescriptor

Parent topic: Descriptor structures

ZPS_tsAplZdpSimpleDescType The ZDP Simple descriptor structure ZPS_tsAplZdpSimpleDescTypeis shown below.

```
typedef struct {
        uint8 u8Endpoint;
        uint16 u16ApplicationProfileId;
        uint16 u16DeviceId;
    union
    {
        ZPS_tsAplZdpSimpleDescBitFields sBitFields;
        uint8 u8Value;
    }uBitUnion;
uint8 u8InClusterCount;
uint16* pu16InClusterList;
uint8 u8OutClusterCount;
uint16* pu16OutClusterList;
}ZPS_tsAplZdpSimpleDescType;
```

where:

- u8Endpointis the number, in the range 1-240, of the endpoint to which the Simple descriptor corresponds.
- u16ApplicationProfileIdis the 16-bit identifier of the ZigBee application profile supported by the endpoint. This must be an application profile identifier issued by the ZigBee Alliance (for Lighting & Occupancy devices, it is 0x0104).
- u16DeviceIdis the 16-bit identifier of the ZigBee device description supported by the endpoint. This must be a device description identifier issued by the ZigBee Alliance.
- sBitFieldsis a structure of type ZPS_tsAplZdpSimpleDescBitFields (described below) containing information about the endpoint.
- u8Valueis used for the union and must be set to 0x00.
- u8InClusterCountis an 8-bit count of the number of input clusters, supported on the endpoint, that will appear in the list pointed to by the pu16InClusterListelement.
- *pu16InClusterListis a pointer to the list of input clusters supported by the endpoint (for use during the service discovery and binding procedures). This is a sequence of 16-bit values, representing the cluster numbers (in the range 1-240), where the number of values is equal to count u8InClusterCount. If this count is zero, the pointer can be set to NULL.
- u8OutClusterCountis an 8-bit count of the number of output clusters, supported on the endpoint, that will appear in the pu16OutClusterListelement.
- *pu16OutClusterListis a pointer to the list of output clusters supported by the endpoint (for use during the service discovery and binding procedures). This is a sequence of 16-bit values, representing the cluster numbers (in the range 1-240), where the number of values is equal to count u8OutClusterCount. If this count is zero, the pointer can be set to NULL.

ZPS_tsAplZdpSimpleDescBitFields The ZPS_tsAplZdpSimpleDescBitFieldsstructure is used by the sBitFields element in the Simple descriptor structure (see above), and is shown below:

```
typedef struct
{
unsigned eDeviceVersion :4;
unsigned eReserved :4;
}ZPS_tsAplZdpSimpleDescBitFields;
```

where:

- eDeviceVersionis a 4-bit value identifying the version of the device description supported by the endpoint.
- eReservedis a 4-bit reserved value.

Parent topic:ZPS_tsAplZdpSimpleDescType

Parent topic: Descriptor structures

Parent topic:ZDP structures

ZDP Request structures These structures are used to represent requests in the ZDP functions. The ZDP request structures are listed below.

Address Discovery Request Structures

1. ZPS_tsAplZdpNwkAddrReq
2. ZPS_tsAplZdpIEEEAddrReq
3. ZPS_tsAplZdpDeviceAnnceReq

   Service Discovery Request Structures

4. ZPS_tsAplZdpNodeDescReq
5. ZPS_tsAplZdpPowerDescReq
6. ZPS_tsAplZdpSimpleDescReq
7. ZPS_tsAplZdpExtendedSimpleDescReq
8. ZPS_tsAplZdpComplexDescReq
9. ZPS_tsAplZdpUserDescReq
10. ZPS_tsAplZdpMatchDescReq
11. ZPS_tsAplZdpActiveEpReq
12. ZPS_tsAplZdpExtendedActiveEpReq
13. ZPS_tsAplZdpUserDescSet
14. ZPS_tsAplZdpSystemServerDiscoveryReq
15. ZPS_tsAplZdpDiscoveryCacheReq
16. ZPS_tsAplZdpDiscoveryStoreReq
17. ZPS_tsAplZdpNodeDescStoreReq
18. ZPS_tsAplZdpPowerDescStoreReq
19. ZPS_tsAplZdpSimpleDescStoreReq
20. ZPS_tsAplZdpActiveEpStoreReq
21. ZPS_tsAplZdpFindNodeCacheReq
22. ZPS_tsAplZdpRemoveNodeCacheReq

    Binding Request Structures

23. ZPS_tsAplZdpEndDeviceBindReq
24. ZPS_tsAplZdpBindUnbindReq
25. ZPS_tsAplZdpBindUnbindReq
26. ZPS_tsAplZdpBindRegisterReq
27. ZPS_tsAplZdpReplaceDeviceReq
28. ZPS_tsAplZdpStoreBkupBindEntryReq
29. ZPS_tsAplZdpRemoveBkupBindEntryReq
30. ZPS_tsAplZdpBackupBindTableReq

31. ZPS_tsAplZdpRecoverBindTableReq
32. ZPS_tsAplZdpBackupSourceBindReq
33. ZPS_tsAplZdpRecoverSourceBindReq

    Network Management Services Request Structures

34. ZPS_tsAplZdpMgmtNwkDiscReq
35. ZPS_tsAplZdpMgmtLqiReq
36. ZPS_tsAplZdpMgmtRtgReq
37. ZPS_tsAplZdpMgmtBindReq
38. ZPS_tsAplZdpMgmtLeaveReq
39. ZPS_tsAplZdpMgmtDirectJoinReq
40. ZPS_tsAplZdpMgmtPermitJoiningReq
41. ZPS_tsAplZdpMgmtCacheReq
42. ZPS_tsAplZdpMgmtNwkUpdateReq
43. ZPS_tsAplZdpParentAnnceReq

ZPS_tsAplZdpNwkAddrReq This structure is used by the function **ZPS_eAplZdpNwkAddrRequest()**. It represents a request for the network address of the node with a given IEEE address.

The ZPS_tsAplZdpNwkAddrReqstructure is detailed below.

```
typedef struct {
  uint64 u64IeeeAddr;
  uint8 u8RequestType;
  uint8 u8StartIndex;
} ZPS_tsAplZdpNwkAddrReq;
```

where:

- u64IeeeAddr is the IEEE address of the node of interest.
- u8RequestType is the type of response required:
  - 0x00: Single device response, which contains only the network address of the target node.
  - 0x01: Extended response, which also includes the network addresses of neighboring nodes.
  - All other values are reserved.
- u8StartIndex is the Neighbor table index of the first neighboring node to be included in the response, if an extended response has been selected.

Parent topic:ZDP Request structures

ZPS_tsAplZdpIEEEAddrReq This structure is used by the function **ZPS_eAplZdpIEEEAddrRequest()**. It represents a request for the IEEE address of a node with a given network address.

The ZPS_tsAplZdpIEEEAddrReqstructure is detailed below.

typedef struct {

uint16 u16NwkAddrOfInterest; uint8 u8RequestType;

uint8 u8StartIndex;

} ZPS_tsAplZdpIEEEAddrReq;

where:

- u16NwkAddrOfInterestis the network address of the node of interest
- u8RequestTypeis the type of response required:
  - 0x00: Single device response, which will contain only the IEEE address of the target node
  - 0x01: Extended response, which will also include the IEEE addresses of neighboring nodes
  - All other values are reserved
- u8StartIndexis the Neighbor table index of the first neighboring node to be included in the response, if an extended response has been selected

Parent topic:ZDP Request structures

ZPS_tsAplZdpDeviceAnnceReq This structure is used by the function **ZPS_eAplZdpDeviceAnnceRequest()**. It represents an announcement that the sending node has joined or rejoined the network.

The ZPS_tsAplZdpDeviceAnnceReqstructure is detailed below.

typedef struct { uint16 u16NwkAddr; uint64 u64IeeeAddr;

uint8 u8Capability;

} ZPS_tsAplZdpDeviceAnnceReq;

where:

- u16NwkAddris the network address of the sending node
- u64IeeeAddris the IEEE address of the sending node
- u8Capabilityis a bitmap representing the capabilities of the sending node. This bitmap is detailed in Table 1 in section ZPS_tsAfNwkJoinIndEvent.

Parent topic:ZDP Request structures

ZPS_tsAplZdpNodeDescReq This structure is used by the function **ZPS_eAplZdpNodeDescRequest()**. It represents a request for the Node descriptor of the node with a given network address.

The ZPS_tsAplZdpNodeDescReqstructure is detailed below.

typedef struct {

uint16 u16NwkAddrOfInterest;

} ZPS_tsAplZdpNodeDescReq;

where u16NwkAddrOfInterestis the network address of the node of interest.

Parent topic:ZDP Request structures

ZPS_tsAplZdpPowerDescReq This structure is used by the function **ZPS_eAplZdpPowerDescRequest()**. It represents a request for the Power descriptor of the node with a given network address.

The ZPS_tsAplZdpPowerDescReqstructure is detailed below.

typedef struct {

uint16 u16NwkAddrOfInterest;

} ZPS_tsAplZdpPowerDescReq;

where u16NwkAddrOfInterestis the network address of the node of interest.

Parent topic:ZDP Request structures

ZPS_tsAplZdpSimpleDescReq This structure is used by the function **ZPS_eAplZdpSimpleDescRequest()**. It represents a request for the Simple descriptor of an endpoint on the node with a given network address.

The ZPS_tsAplZdpSimpleDescReqstructure is detailed below.

```
typedef struct {
    uint16 u16NwkAddrOfInterest;
    uint8 u8EndPoint;
} ZPS_tsAplZdpSimpleDescReq;
```

where:

- u16NwkAddrOfInterestis the network address of the node of interest.
- u8EndPointis the number of the relevant endpoint on the node (1-240).

Parent topic:ZDP Request structures

ZPS_tsAplZdpExtendedSimpleDescReq This structure is used by the **ZPS_eAplZdpExtendedSimpleDescRequest()**function. It represents a request for the Simple descriptor of an endpoint on the node with a given network address. This request is required when the endpoint has more input/output clusters than the usual **ZPS_eAplZdpSimpleDescRequest()** function can deal with.

The ZPS_tsAplZdpExtendedSimpleDescReqstructure is detailed below.

typedef struct { uint16 u16NwkAddr; uint8 u8EndPoint;

uint8 u8StartIndex;

} ZPS_tsAplZdpExtendedSimpleDescReq;

where:

- u16NwkAddrOfInterestis the network address of the node of interest
- u8EndPointis the number of the relevant endpoint on the node (1-240)
- u8StartIndexis the index of the first cluster of interest in the input and output cluster lists for the endpoint (this and subsequent clusters will be reported in the response)

Parent topic:ZDP Request structures

ZPS_tsAplZdpComplexDescReq This structure is used by the function **ZPS_eAplZdpComplexDescRequest()**. It represents a request for the Complex descriptor of the node with a given network address.

The ZPS_tsAplZdpComplexDescReqstructure is detailed below.

typedef struct {

uint16 u16NwkAddrOfInterest;

} ZPS_tsAplZdpComplexDescReq;

where u16NwkAddrOfInterestis the network address of the node of interest.

Parent topic:ZDP Request structures

ZPS_tsAplZdpUserDescReq This structure is used by the function **ZPS_eAplZdpUserDescRequest()**. It represents a request for the User descriptor of the node with a given network address.

The ZPS_tsAplZdpUserDescReqstructure is detailed below.

typedef struct {

uint16 u16NwkAddrOfInterest;

} ZPS_tsAplZdpUserDescReq;

where u16NwkAddrOfInterestis the network address of the node of interest.

Parent topic:ZDP Request structures

ZPS_tsAplZdpMatchDescReq This structure is used by the function **ZPS_eAplZdpMatchDescRequest()**. It represents a request for nodes with endpoints that match certain criteria in their Simple descriptors.

The ZPS_tsAplZdpMatchDescReqstructure is detailed below.

typedef struct {

uint16 u16NwkAddrOfInterest; uint16 u16ProfileId;

/* rest of message is variable length */ uint8 u8NumInClusters;

uint16* pu16InClusterList; uint8 u8NumOutClusters; uint16* pu16OutClusterList;

} ZPS_tsAplZdpMatchDescReq;

where:

- u16NwkAddrOfInterestis the network address of the node of interest
- u16ProfileIdis the identifier of the ZigBee application profile used
- u8NumInClustersis the number of input clusters to be matched
- pu16InClusterListis a pointer to the list of input clusters to be matched - this is a variable-length list of input cluster IDs, two bytes for each cluster
- u8NumOutClustersis the number of output clusters to be matched
- pu16OutClusterListis a pointer to the list of output clusters to be matched - this is a variable-length list of output cluster IDs, two bytes for each cluster

Parent topic:ZDP Request structures

ZPS_tsAplZdpActiveEpReq This structure is used by the function **ZPS_eAplZdpActiveEpRequest()**. It represents a request for a list of the active endpoints on the node with a given network address.

The ZPS_tsAplZdpActiveEpReqstructure is detailed below.

typedef struct {

uint16 u16NwkAddrOfInterest;

} ZPS_tsAplZdpActiveEpReq;

where u16NwkAddrOfInterestis the network address of the node of interest.

Parent topic:ZDP Request structures

ZPS_tsAplZdpExtendedActiveEpReq This structure is used by the function **ZPS_eAplZdpExtendedActiveEpRequest()**. It represents a request for a list of the active endpoints on the node with a given network address. This request is required when the node has more active endpoints than the usual **ZPS_eAplZdpActiveEpRequest()** function can deal with.

The ZPS_tsAplZdpExtendedActiveEpReqstructure is detailed below.

typedef struct { uint16 u16NwkAddr;

uint8 u8StartIndex;

} ZPS_tsAplZdpExtendedActiveEpReq;

where:

- u16NwkAddris the network address of the node of interest
- u8StartIndexis the index of the first endpoint of interest in the list of active endpoints (this and subsequent endpoints will be reported in the response)

Parent topic:ZDP Request structures

ZPS_tsAplZdpUserDescSet This structure is used by the function **ZPS_eAplZdpUserDescSetRequest()**. It represents a request used to configure the User descriptor on a remote node.

The ZPS_tsAplZdpUserDescSetstructure is detailed below.

typedef struct {

uint16 u16NwkAddrOfInterest; uint8 u8Length;

char szUserDescriptor[ZPS_ZDP_LENGTH_OF_USER_DESC];

} ZPS_tsAplZdpUserDescSet;

where:

- u16NwkAddrOfInterestis the network address of the node of interest
- u8Lengthis the length of the User descriptor
- szUserDescriptoris the new User descriptor for the remote node as a character array.

Parent topic:ZDP Request structures

ZPS_tsAplZdpSystemServerDiscoveryReq This structure is used by the **ZPS_eAplZdpSystemServerDiscoveryRequest()** function. It represents a request for information on the available services of a remote node.

The ZPS_tsAplZdpSystemServerDiscoveryReqstructure is detailed below.

```
typedef struct {
uint16 u16ServerMask;
} ZPS_tsAplZdpSystemServerDiscoveryReq;
```

where u16ServerMaskis a bitmask representing the required services (1 for 'required', 0 for 'not required'). This bitmask is detailed in the table below.

Services Bitmask

Bit	Service
0	Primary Trust Centre
1	Backup Trust Centre
2	Primary Binding Table Cache
3	Backup Binding Table Cache
4	Primary Discovery Cache
5	Back-up Discovery Cache
6	Network Manager
7-15	Reserved

Parent topic:ZDP Request structures

ZPS_tsAplZdpDiscoveryCacheReq This structure is used by the function **ZPS_eAplZdpDiscoveryCacheRequest()**. It represents a request to find the nodes in the network which have a primary discovery cache.

The ZPS_tsAplZdpDiscoveryCacheReqstructure is detailed below.

typedef struct { uint16 u16NwkAddr; uint64 u64IeeeAddr;

} ZPS_tsAplZdpDiscoveryCacheReq;

where:

- u16NwkAddris the network address of the sending node
- u64IeeeAddris the IEEE address of the sending node

Parent topic:ZDP Request structures

ZPS_tsAplZdpDiscoveryStoreReq This structure is used by the function **ZPS_eAplZdpDiscoveryStoreRequest()**. It represents a request to a remote node to reserve memory space to store the local node's 'discovery information'.

The ZPS_tsAplZdpDiscoveryStoreReqstructure is detailed below.

typedef struct { uint16 u16NwkAddr; uint64 u64IeeeAddr;

uint8 u8NodeDescSize; uint8 u8PowerDescSize; uint8 u8ActiveEpSize; uint8 u8SimpleDescCount;

/* Rest of message is variable length */ uint8* pu8SimpleDescSizeList;

} ZPS_tsAplZdpDiscoveryStoreReq;

where:

- u16NwkAddris the network address of the sending node
- u64IeeeAddris the IEEE address of the sending node
- u8NodeDescSizeis the size of the Node descriptor to store
- u8PowerDescSizeis the size of the Power descriptor to store
- u8ActiveEpSizeis the size of the list of active endpoints to store
- u8SimpleDescCountis the number of Simple descriptors to store
- pu8SimpleDescSizeListis a pointer to a list of sizes of the Simple descriptors

Parent topic:ZDP Request structures

ZPS_tsAplZdpNodeDescStoreReq This structure is used by the function **ZPS_eAplZdpNodeDescStoreRequest()**. It represents a request to a remote node to store the Node descriptor of the local node.

The ZPS_tsAplZdpNodeDescStoreReqstructure is detailed below.

typedef struct { uint16 u16NwkAddr; uint64 u64IeeeAddr;

/* Rest of message is variable length */ ZPS_tsAplZdpNodeDescriptor sNodeDescriptor;

} ZPS_tsAplZdpNodeDescStoreReq;

where:

- u16NwkAddris the network address of the sending node
- u64IeeeAddris the IEEE address of the sending node
- sNodeDescriptoris a pointer to the Node descriptor to store (this is itself a structure of the type ZPS_tsAplZdpNodeDescriptor, detailed in Section8.2.1.1)

Parent topic:ZDP Request structures

ZPS_tsAplZdpPowerDescStoreReq This structure is used by the function **ZPS_eAplZdpPowerDescStoreRequest()**. It represents a request to a remote node to store the Power descriptor of the local node.

The ZPS_tsAplZdpPowerDescStoreReqstructure is detailed below.

typedef struct { uint16 u16NwkAddr; uint64 u64IeeeAddr;

/* Rest of message is variable length */ ZPS_tsAplZdpNodePowerDescriptor sPowerDescriptor;

} ZPS_tsAplZdpPowerDescStoreReq;

where:

- u16NwkAddris the network address of the sending node
- u64IeeeAddris the IEEE address of the sending node
- sPowerDescriptoris a pointer to the Power descriptor to store (this is itself a structure of the type ZPS_tsAplZdpNodePowerDescriptor, detailed in Section8.2.1.2)

Parent topic:ZDP Request structures

ZPS_tsAplZdpSimpleDescStoreReq This structure is used by the function **ZPS_eAplZdpSimpleDescStoreRequest()**. It represents a request to a remote node to store the Simple descriptor of one of the local node's endpoints.

The ZPS_tsAplZdpSimpleDescStoreReqstructure is detailed below.

typedef struct { uint16 u16NwkAddr; uint64 u64IeeeAddr; uint8 u8Length;

/* Rest of message is variable length */ ZPS_tsAplZdpSimpleDescType sSimpleDescriptor;

} ZPS_tsAplZdpSimpleDescStoreReq;

where:

- u16NwkAddris the network address of the sending node
- u64IeeeAddris the IEEE address of the sending node
- u8Lengthis the length of the Simple descriptor to store
- sSimpleDescriptoris a pointer to the Simple descriptor to store (this is itself a structure of the type ZPS_tsAplZdpSimpleDescType, detailed in Section8.2.1.3)

Parent topic:ZDP Request structures

ZPS_tsAplZdpActiveEpStoreReq This structure is used by the function **ZPS_eAplZdpActiveEpStoreRequest()**. It represents a request to a remote node to store the list of active endpoints of the local node.

The ZPS_tsAplZdpActiveEpStoreReqstructure is detailed below.

typedef struct { uint16 u16NwkAddr; uint64 u64IeeeAddr;

uint8 u8ActiveEPCount;

/* Rest of message is variable length */ uint8* pu8ActiveEpList;

} ZPS_tsAplZdpActiveEpStoreReq;

where:

- u16NwkAddris the network address of the sending node
- u64IeeeAddris the IEEE address of the sending node
- u8ActiveEPCountis the number of active endpoints in the list to store
- pu8ActiveEpListis a pointer to the list of active endpoints to store

Parent topic:ZDP Request structures

ZPS_tsAplZdpFindNodeCacheReq This structure is used by the function **ZPS_eAplZdpActiveEpStoreRequest()**. It represents a request to search for nodes in the network that hold 'discovery information' about a particular node.

The ZPS_tsAplZdpFindNodeCacheReqstructure is detailed below.

```
typedef struct { uint16 u16NwkAddr; uint64 u64IeeeAddr;
} ZPS_tsAplZdpFindNodeCacheReq;
```

where:

- u16NwkAddris the network address of the node of interest
- u64IeeeAddris the IEEE address of the node of interest

Parent topic:ZDP Request structures

ZPS_tsAplZdpRemoveNodeCacheReq This structure is used by the function **ZPS_eAplZdpActiveEpStoreRequest()**. It represents a request to a remote node to remove from its Primary Discovery Cache all 'discovery information' relating to a particular End Device.

The ZPS_tsAplZdpRemoveNodeCacheReqstructure is detailed below.

```
typedef struct { uint16 u16NwkAddr; uint64 u64IeeeAddr;
} ZPS_tsAplZdpRemoveNodeCacheReq;
```

where:

- u16NwkAddris the network address of the End Device of interest
- u64IeeeAddris the IEEE address of the End Device of interest

Parent topic:ZDP Request structures

ZPS_tsAplZdpEndDeviceBindReq This structure is used by the function **ZPS_eAplZdpEndDeviceBindRequest()**. It represents a request to the Coordinator to bind an endpoint on the local node to an endpoint on a remote node (the Coordinator must match two such binding requests, from the local node and remote node).

The ZPS_tsAplZdpEndDeviceBindReqstructure is detailed below.

```
typedef struct {
uint16 u16BindingTarget;
uint64 u64SrcIeeeAddress;
uint8 u8SrcEndpoint;
uint16 u16ProfileId;
/* Rest of the message is variable length */
uint8 u8NumInClusters;
uint16 *pu16InClusterList;
uint8 u8NumOutClusters;
uint16 *pu16OutClusterList;
} ZPS_tsAplZdpEndDeviceBindReq;
```

where:

- u16BindingTargetis the network address of the node to hold the binding (either a node with primary binding table cache or the local node).
- u64SrcIeeeAddressis the IEEE address of the local node.
- u8SrcEndpointis the number of the local endpoint to be bound (1-240).
- u16ProfileIdis the application profile ID to be matched for the binding.
- u8NumInClustersis the number of input clusters of the local endpoint (available for matching with output clusters of remote node to be bound).
- pu16InClusterListis a pointer to the input cluster list of the local endpoint (containing clusters for matching with output clusters of remote node).
- u8NumOutClustersis the number of output clusters of the local endpoint (available for matching with input clusters of remote node to be bound).
- pu16OutClusterListis a pointer to the output cluster list of the local endpoint (containing clusters for matching with input clusters of remote node).

Parent topic:ZDP Request structures

ZPS_tsAplZdpBindUnbindReq This structure is used by the function **ZPS_eAplZdpBindUnbindRequest()**. It represents a request for a modification of the Binding table on the target node, in order to either bind or unbind two nodes in the network.

The ZPS_tsAplZdpBindUnbindReqstructure is detailed below.

```
typedef struct {
uint64 u64SrcAddress;
uint8 u8SrcEndpoint;
uint16 u16ClusterId;
uint8 u8DstAddrMode;
union {
  struct {
      uint16 u16DstAddress;
        } sShort;
  struct {
      uint64 u64DstAddress;
      uint8 u8DstEndPoint;
        } sExtended;
  } uAddressField;
} ZPS_tsAplZdpBindUnbindReq;
```

where:

- u64SrcAddressis the IEEE address of the source node for the binding
- u8SrcEndpointis the number of the source endpoint for the binding (1-240)
- u16ClusterIdis the ID of the cluster (on the local endpoint) for the binding

- u8DstAddrModeis the destination addressing mode (see Table 14 below):
  - ZPS_E_ADDR_MODE_SHORT: network address (u8DstEndPointis unspecified)
  - ZPS_E_ADDR_MODE_IEEE: IEEE address (u8DstEndPointis specified)
  - All other values are reserved
- u16DstAddressor u64DstAddressis the address of the destination node for the binding:
  - network address u16DstAddressif u8DstAddrModeis set to ZPS_E_ADDR_MODE_SHORT
  - IEEE address u64DstAddressif 8DstAddrModeis set to ZPS_E_ADDR_MODE_IEEE
- u8DstEndPointis the number of the destination endpoint for the binding

(1-240) - not required if u8DstAddrModeset to ZPS_E_ADDR_MODE_SHORT (network address)

Addressing modes

u8DstAddrMode	Code	Description
0x02	ZPS_E_ADDR_MODE_SHORT	16-bit Network (Short) address
0x03	ZPS_E_ADDR_MODE_IEEE	64-bit IEEE/MAC address

Parent topic:ZDP Request structures

ZPS_tsAplZdpBindRegisterReq This structure is used by the function **ZPS_eAplZdpBindRegisterRequest()**. It represents a request to inform a remote node with a primary binding table cache that the local node will hold its own Binding table entries.

The ZPS_tsAplZdpBindRegisterReqstructure is detailed below.

```
typedef struct {
uint64 u64NodeAddress;
} ZPS_tsAplZdpBindRegisterReq;
```

where u64NodeAddressis the IEEE address of the local node.

Parent topic:ZDP Request structures

ZPS_tsAplZdpReplaceDeviceReq This structure is used by the function **ZPS_eAplZdpReplaceDeviceRequest()**. It represents a request to a remote node (with a primary binding table cache) to modify its binding table entries by replacing an IEEE address and/or associated endpoint number.

The ZPS_tsAplZdpReplaceDeviceReqstructure is detailed below.

```
typedef struct {
uint64 u64OldAddress; uint8 u8OldEndPoint; uint64 u64NewAddress; uint8 u8NewEndPoint;
} ZPS_tsAplZdpReplaceDeviceReq;
```

where:

- u64OldAddressis the IEEE address to be replaced
- u8OldEndPointis the endpoint number to be replaced

(0-240, where 0 indicates that the endpoint number is not to be replaced)

- u64NewAddressis the replacement IEEE address
- u8NewEndPointis the replacement endpoint number (1-240)

Parent topic:ZDP Request structures

ZPS_tsAplZdpStoreBkupBindEntryReq This structure is used by the function **ZPS_eAplZdpStoreBkupBindEntryRequest()**. It represents a request to a remote node to save a back-up of an entry from the local primary binding table cache.

The ZPS_tsAplZdpStoreBkupBindEntryReqstructure is detailed below.

```
typedef struct {
uint64 u64SrcAddress;
uint8 u8SrcEndPoint;
uint16 u16ClusterId;
uint8 u8DstAddrMode;
union {
    struct {
        uint16 u16DstAddress;
            } sShort;
     struct {
        uint64 u64DstAddress;
        uint8 u8DstEndPoint;
    } sExtended;
     };
} ZPS_tsAplZdpStoreBkupBindEntryReq;
```

where:

- u64SrcAddressis the IEEE address of the source node for the binding entry
- u8SrcEndpointis the number of the source endpoint for the binding (1-240)
- u16ClusterIdis the ID of the cluster (on the local endpoint) for the binding
- u8DstAddrModeis the destination addressing mode for remaining elements (see Table 15 below)
- u16DstAddressis the address of the destination node for the binding (address type according to setting of u8DstAddrMode)
- u8DstEndPointis the number of the destination endpoint for the binding (1-240)

Addressing modes

u8DstAddrMode	Code	Description
0x01	ZPS_E_ADDR_MODE_GROUP	16-bit Group address
0x03	ZPS_E_ADDR_MODE_IEEE	64-bit IEEE/MAC address

Parent topic:ZDP Request structures

ZPS_tsAplZdpRemoveBkupBindEntryReq This structure is used by the **ZPS_eAplZdpRemoveBkupBindEntryRequest()** function. It represents a request to a remote node to remove the back-up of an entry from the local primary binding table cache.

The ZPS_tsAplZdpRemoveBkupBindEntryReqstructure is detailed below.

```
typedef struct {
uint64 u64SrcAddress;
uint8 u8SrcEndPoint;
uint16 u16ClusterId;
uint8 u8DstAddrMode;
 union {
    struct {
      uint16 u16DstAddress;
      } sShort;
    struct
```

(continues on next page)

(continued from previous page)

```
        {
      uint64 u64DstAddress;
      uint8 u8DstEndPoint;
      } sExtended;
        };
} ZPS_tsAplZdpRemoveBkupBindEntryReq;
```

where:

- u64SrcAddressis the IEEE address of the source node for the binding entry.
- u8SrcEndpointis the number of the source endpoint for the binding (1-240).
- u16ClusterIdis the ID of the cluster (on the local endpoint) for the binding.
- u8DstAddrModeis the destination addressing mode for remaining elements (see the Table below) .
- u16DstAddressis the address the destination node for the binding (address type according to setting of u8DstAddrMode).
- u8DstEndPointis the number of the destination endpoint for the binding (1-240).

Addressing modes

u8DstAddrMode	Code	Description
0x01	ZPS_E_ADDR_MODE_GROUP	16-bit Group address
0x03	ZPS_E_ADDR_MODE_IEEE	64-bit IEEE/MAC address

Parent topic:ZDP Request structures

ZPS_tsAplZdpBackupBindTableReq This structure is used by the function **ZPS_eAplZdpBackupBindTableRequest()**. It represents a request to a remote node to save a back-up of the local primary binding table cache (whole or in part).

The ZPS_tsAplZdpBackupBindTableReqstructure is detailed below.

```
typedef struct {
    uint16 u16BindingTableEntries;
    uint16 u16StartIndex;
    uint16 u16BindingTableListCount;
    /* Rest of message is variable length */
    ZPS_tsAplZdpBindingTable sBindingTable;
} ZPS_tsAplZdpBackupBindTableReq;
```

where:

- u16BindingTableEntriesis the total number of entries in the primary binding table cache.
- u16StartIndexis the binding table index of the first entry to be backed up.
- u16BindingTableListCountis the number of binding table entries in the list to be backed up (sBindingTable).
- sBindingTableis a pointer to the list of binding table entries to be backed up. Each list item is of the type ZPS_tsAplZdpBindingTabledetailed below:

ZPS_tsAplZdpBindingTable

```
typedef struct
{
uint64 u64SourceAddress;
```

(continues on next page)

(continued from previous page)

```
ZPS_tsAplZdpBindingTableEntry* psBindingTableEntryForSpSrcAddr;
}ZPS_tsAplZdpBindingTable;
```

where:

- u64SourceAddressis the IEEE source address for the binding table entry.
- psBindingTableEntryForSpSrcAddris the binding table entry. This is of the type ZPS_tsAplZdpBindingTableEntrydetailed below.

ZPS_tsAplZdpBindingTableEntry

```
typedef struct
{
    uint16 u16ClusterId;
    uint8 u8SourceEndpoint;
    uint8 u8DstAddrMode;
union {
     struct {
        uint16 u16DstAddress;
      } sShort;
     struct {
        uint64 u64DstAddress;
        uint8 u8DstEndPoint;
      } sExtended;
    };
}ZPS_tsAplZdpBindingTableEntry;
```

where:

- u16ClusterIdis the ID of the cluster (on the local endpoint) for the binding
- u8SrcEndpointis the number of the source endpoint for the binding (1-240)
- u8DstAddrModeis the destination addressing mode for remaining elements (see Table below)
- u16DstAddressis the address the destination node for the binding (address type according to setting of u8DstAddrMode)
- u8DstEndPointis the number of the destination endpoint for the binding (1-240)

Addressing modes

u8DstAddrMode	Code	Description
0x01	ZPS_E_ADDR_MODE_GROUP	16-bit Group address
0x03	ZPS_E_ADDR_MODE_IEEE	64-bit IEEE/MAC address

Parent topic:ZDP Request structures

ZPS_tsAplZdpRecoverBindTableReq This structure is used by the function **ZPS_eAplZdpRecoverBindTableRequest()**. It represents a request to a remote node to recover a back-up of the local primary binding table cache.

The ZPS_tsAplZdpRecoverBindTableReqstructure is detailed below.

```
typedef struct {
uint16 u16StartIndex;
} ZPS_tsAplZdpRecoverBindTableReq;
```

where u16StartIndexis the binding table index of the first entry to be recovered.

Parent topic:ZDP Request structures

ZPS_tsAplZdpBackupSourceBindReq This structure is used by the function **ZPS_eAplZdpBackupSourceBindRequest()**. It represents a request to a remote node to save a back-up of the local node's source binding table (whole or in part).

The ZPS_tsAplZdpBackupSourceBindReqstructure is detailed below.

```
typedef struct {
  uint16 u16SourceTableEntries;
  uint16 u16StartIndex;
  uint16 u16SourceTableListCount;
  /* Rest of message is variable length */
  uint64* pu64SourceAddress;
} ZPS_tsAplZdpBackupSourceBindReq;
```

where:

- u16SourceTableEntriesis the total number of entries in the source binding table.
- u16StartIndexis the binding table index of the first entry to be backed up.
- u16SourceTableListCountis the number of binding table entries in the list to be backed up (pu64SourceAddress).
- pu64SourceAddressis a pointer to the list of IEEE source addresses corresponding to the binding table entries to be backed up.

Parent topic:ZDP Request structures

ZPS_tsAplZdpRecoverSourceBindReq This structure is used by the function **ZPS_eAplZdpRecoverSourceBindRequest()**. It represents a request to a remote node to recover the back-up of the local node's source binding table (whole or in part).

The ZPS_tsAplZdpRecoverSourceBindReqstructure is detailed below.

```
typedef struct {
uint16 u16StartIndex;
} ZPS_tsAplZdpRecoverSourceBindReq;
```

where u16StartIndexis the binding table index of the first entry to be recovered.

Parent topic:ZDP Request structures

ZPS_tsAplZdpMgmtNwkDiscReq This structure is used by the function **ZPS_eAplZdpMgmtNwkDiscRequest()**. It represents a request to a remote node to discover any other wireless networks that are operating in the neighborhood.

The ZPS_tsAplZdpMgmtNwkDiscReqstructure is detailed below.

```
typedef struct {
 uint32 u32ScanChannels;
 uint8 u8ScanDuration;
 uint8 u8StartIndex;
} ZPS_tsAplZdpMgmtNwkDiscReq;
```

where:

- u32ScanChannelsis a bitmask of the radio channels to scan ('1' means scan, '0' means do not scan):

  - Bits 0 to 26 respectively represent channels 0 to 26 (only bits 11 to 26 are relevant to the 2400-MHz band)
  - Bits 27 to 31 are reserved

- u8ScanDurationis a value in the range 0x00 to 0x0E that determines the time spent scanning each channel - this time is proportional to 2u8ScanDuration+1
- u8StartIndexis the index of the first result from the results list to include in the response to this request

Parent topic:ZDP Request structures

ZPS_tsAplZdpMgmtLqiReq This structure is used by the function **ZPS_eAplZdpMgmtLqiRequest()**. It represents a request to a remote node to provide a list of neighboring nodes, from its Neighbor table, including a radio signal strength (LQI) value for each of these nodes.

The ZPS_tsAplZdpMgmtLqiReqstructure is detailed below.

```
typedef struct {
uint8 u8StartIndex;
} ZPS_tsAplZdpMgmtLqiReq;
```

where u8StartIndexis the Neighbor table index of the first entry to be included in the response to this request.

Parent topic:ZDP Request structures

ZPS_tsAplZdpMgmtRtgReq This structure is used by the function **ZPS_eAplZdpMgmtRtgRequest()**. It represents a request to a remote node to provide the contents of its Routing table.

The ZPS_tsAplZdpMgmtRtgReqstructure is detailed below.

```
typedef struct {
uint8 u8StartIndex;
} ZPS_tsAplZdpMgmtRtgReq;
```

where u8StartIndexis the Routing table index of the first entry to be included in the response to this request.

Parent topic:ZDP Request structures

ZPS_tsAplZdpMgmtBindReq This structure is used by the function **ZPS_eAplZdpMgmtBindRequest()**. It represents a request to a remote node to provide the contents of its Binding table.

The ZPS_tsAplZdpMgmtBindReqstructure is detailed below.

```
typedef struct {
    uint8 u8StartIndex;
} ZPS_tsAplZdpMgmtBindReq;
```

where u8StartIndexis the Binding table index of the first entry to be included in the response to this request.

Parent topic:ZDP Request structures

ZPS_tsAplZdpMgmtLeaveReq This structure is used by the function **ZPS_eAplZdpMgmtLeaveRequest()**. It requests a remote node to leave the network.

The ZPS_tsAplZdpMgmtLeaveReqstructure is detailed below.

```
typedef struct {
    uint64 u64DeviceAddress;
    uint8 u8Flags;
} ZPS_tsAplZdpMgmtLeaveReq;
```

where:

- u64DeviceAddressis the IEEE address of the device being asked to leave the network.
- u8Flagsis an 8-bit bitmap containing the following flags:
  - Rejoin flag (bit 0): Set to 1 if the node requested to leave the network should immediately try to rejoin the network, otherwise set to 0.
  - Remove Children flag (bit 1): Set to 1 if the node requested to leave the network should also request its own children (if any) to leave the network, otherwise set to 0.
  - Reserved (bits 7-2).

Parent topic:ZDP Request structures

ZPS_tsAplZdpMgmtDirectJoinReq This structure is used by the function **ZPS_eAplZdpMgmtDirectJoinRequest()**. It requests a remote node to allow a particular device to join it (and therefore the network).

The ZPS_tsAplZdpMgmtDirectJoinReqstructure is detailed below.

```
typedef struct { uint64 u64DeviceAddress; uint8 u8Capability; } ZPS_tsAplZdpMgmtDirectJoinReq;
```

where:

- u64DeviceAddressis the IEEE address of the device to be allowed to join
- u8Capabilityis a bitmask of the operating capabilities of the device to be allowed to join. This bitmap is detailed in Table in section ZPS_tsAfNwkJoinIndEvent.

Parent topic:ZDP Request structures

ZPS_tsAplZdpMgmtPermitJoiningReq This structure is used by the function **ZPS_eAplZdpMgmtPermitJoiningRequest()**. It requests a remote node (Router or Coordinator) to enable or disable joining for a specified amount of time.

The ZPS_tsAplZdpMgmtPermitJoiningReqstructure is detailed below.

typedef struct {

uint8 u8PermitDuration; bool_t bTcSignificance;

} ZPS_tsAplZdpMgmtPermitJoiningReq;

where:

- u8PermitDurationis the time period, in seconds, during which joining will be allowed (0x00 means that joining is enabled or disabled with no time limit)
- bTcSignificancedetermines whether the remote device is a 'Trust Centre':
  - TRUE: A Trust Centre
  - FALSE: Not a Trust Centre

Parent topic:ZDP Request structures

ZPS_tsAplZdpMgmtCacheReq This structure is used by the function **ZPS_eAplZdpMgmtCacheRequest()**. It requests a remote node to provide a list of the End Devices registered in its primary discovery cache.

The ZPS_tsAplZdpMgmtCacheReqstructure is detailed below.

typedef struct {

uint8 u8StartIndex;

} ZPS_tsAplZdpMgmtCacheReq;

where u8StartIndexis the discovery cache index of the first entry to be included in the response to this request.

Parent topic:ZDP Request structures

ZPS_tsAplZdpMgmtNwkUpdateReq This structure is used by the function **ZPS_eAplZdpMgmtNwkUpdateRequest()**. It requests an update of network parameters related to radio communication and may optionally initiate an energy scan in the 2400-MHz band.

The ZPS_tsAplZdpMgmtNwkUpdateReqstructure is detailed below.

typedef struct {

uint32 u32ScanChannels; uint8 u8ScanDuration; uint8 u8ScanCount; uint8 u8NwkUpdateId;

uint16 u16NwkManagerAddr;

} ZPS_tsAplZdpMgmtNwkUpdateReq;

where:

- u32ScanChannelsis a bitmask of the radio channels to be scanned ('1' means scan, '0' means do not scan):
  - Bits 0 to 26 respectively represent channels 0 to 26 (only bits 11 to 26 are relevant to the 2400-MHz band)
  - Bits 27 to 31 are reserved
- u8ScanDurationis a key value used to determine the action to be taken, as follows:
  - 0x00-0x05: Indicates that an energy scan is required and determines the time to be spent scanning each channel - this time is proportional to 2u8ScanDuration+1. The set of channels to scan is specified through

u32ScanChannelsand the maximum number of scans is equal to the value of u8ScanCount. Valid for unicasts only

- 0x06-0xFD: Reserved
- 0xFE: Indicates that radio channel is to be changed to single channel specified through u32ScanChannelsand that network manager address to be set to that specified through u16NwkManagerAddr. Valid for broadcasts only
- 0xFF: Indicates that stored radio channel mask to be updated with that specified through u32ScanChannels(but scan not required). Valid for broadcasts only.
- u8ScanCountis the number of energy scans to be conducted and reported. Valid only if a scan has been enabled through u8ScanDuration(0x00-0x05)
- u8NwkUpdateIdis a value set by the Network Channel Manager before the request is sent. Valid only if u8ScanDurationset to 0xFE or 0xFF
- u16NwkManagerAddris the 16-bit network address of the Network Manager (node nominated to manage radio-band operation of network). Valid only if u8ScanDurationset to 0xFF

Parent topic:ZDP Request structures

ZPS_tsAplZdpParentAnnceReq This structure is used by the function **ZPS_eAplZdpParentAnnceReq()**, which sends out a Parent Announcement message. The structure specifies the nodes that are the children of the local node which called the function.

The ZPS_tsAplZdpParentAnnceReqstructure is detailed below.

```
typedef struct {
uint8 u8NumberOfChildren; uint64* pu64ChildList;
} ZPS_tsAplZdpParentAnnceReq;
```

where:

- u8NumberOfChildrenis the number of child nodes
- pu64ChildListis a pointer to a list of the 64-bit IEEE/MAC addresses of the child nodes

Parent topic:ZDP Request structures

Parent topic:ZDP structures

ZDP response structures This section details the structures that are used to store ZDP responses, resulting from requests sent using the ZDP functions. A received response is collected using the function **ZQ_bZQueueReceive()**. As part of this function call, you must provide a pointer to a structure to store the message data. This structure must be of the appropriate type for the response, from those described in this section.

The ZDP response structures are listed below.

1. ZPS_tsAplZdpNwkAddrRsp
2. ZPS_tsAplZdpIeeeAddrRsp

   Service Discovery Response Structures

3. ZPS_tsAplZdpNodeDescRsp
4. ZPS_tsAplZdpPowerDescRsp
5. ZPS_tsAplZdpSimpleDescRsp
6. ZPS_tsAplZdpExtendedSimpleDescRsp
7. ZPS_tsAplZdpComplexDescRsp
8. ZPS_tsAplZdpUserDescRsp
9. ZPS_tsAplZdpMatchDescRsp
10. ZPS_tsAplZdpActiveEpRsp
11. ZPS_tsAplZdpExtendedActiveEpRsp
12. ZPS_tsAplZdpUserDescConf
13. ZPS_tsAplZdpSystemServerDiscoveryRsp
14. ZPS_tsAplZdpDiscoveryCacheRsp
15. ZPS_tsAplZdpDiscoveryStoreRsp
16. ZPS_tsAplZdpNodeDescStoreRsp
17. ZPS_tsAplZdpPowerDescStoreRsp
18. ZPS_tsAplZdpSimpleDescStoreRsp
19. ZPS_tsAplZdpActiveEpStoreRsp
20. ZPS_tsAplZdpFindNodeCacheRsp

21. ZPS_tsAplZdpRemoveNodeCacheRsp

    Binding Response Structures

22. ZPS_tsAplZdpEndDeviceBindRsp
23. ZPS_tsAplZdpBindRsp
24. ZPS_tsAplZdpUnbindRsp
25. ZPS_tsAplZdpBindRegisterRsp
26. ZPS_tsAplZdpReplaceDeviceRsp
27. ZPS_tsAplZdpStoreBkupBindEntryRsp
28. ZPS_tsAplZdpRemoveBkupBindEntryRsp
29. ZPS_tsAplZdpBackupBindTableRsp
30. ZPS_tsAplZdpRecoverBindTableRsp
31. ZPS_tsAplZdpBackupSourceBindRsp
32. ZPS_tsAplZdpRecoverSourceBindRsp

    Network Management Services Response Structures

33. ZPS_tsAplZdpMgmtNwkDiscRsp
34. ZPS_tsAplZdpMgmtLqiRsp
35. ZPS_tsAplZdpMgmtRtgRsp
36. ZPS_tsAplZdpMgmtBindRsp
37. ZPS_tsAplZdpMgmtLeaveRsp
38. ZPS_tsAplZdpMgmtDirectJoinRsp
39. ZPS_tsAplZdpMgmtPermitJoiningRsp
40. ZPS_tsAplZdpMgmtCacheRsp
41. ZPS_tsAplZdpMgmtNwkUpdateNotify
42. ZPS_tsAplZdpParentAnnceRsp

ZPS_tsAplZdpNwkAddrRsp This structure is used to store NWK_addr_rsp message data - a response to a call to the function **ZPS_eAplZdpNwkAddrRequest()**. This response contains the network address of the node with a given IEEE address.

The ZPS_tsAplZdpNwkAddrRspstructure is detailed below.

```
typedef struct { uint8 u8Status;
    uint64 u64IeeeAddrRemoteDev;
    uint16 u16NwkAddrRemoteDev;
    uint8 u8NumAssocDev;
    uint8 u8StartIndex;
    /* Rest of the message is variable Length */
    uint16* pNwkAddrAssocDevList;
} ZPS_tsAplZdpNwkAddrRsp;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpNwkAddrRequest()**
- u64IeeeAddrRemoteDevis the IEEE address of the remote node that sent the response (this is the IEEE address specified in the original request)
- u16NwkAddrRemoteDevis the network address of the remote node that sent the response (this is the network address that was requested)

- u8NumAssocDevis the number of neighboring nodes for which network addresses are also being reported (in the remainder of the structure)
- u8StartIndexis the index in the remote node's Neighbor table of the first entry to be included in this report. This element should be ignored if the element u8NumAssocDevis 0.
- pNwkAddrAssocDevListis a pointer to a list of 16-bit network addresses of the remote node's neighbors (this is a variable-length list with four bytes per node). This element should be ignored if the element u8NumAssocDevis 0.

Parent topic:ZDP response structures

ZPS_tsAplZdpIeeeAddrRsp This structure is used to store IEEE_addr_rsp message data - a response to a call to the function **ZPS_eAplZdpIeeeAddrRequest()**. This response contains the IEEE address of the node with a given network address.

The ZPS_tsAplZdpIeeeAddrRspstructure is detailed below.

```
typedef struct
    { uint8 u8Status;
     uint64 u64IeeeAddrRemoteDev;
     uint16 u16NwkAddrRemoteDev;
     uint8 u8NumAssocDev;
     uint8 u8StartIndex;
    /* Rest of the message is variable Length */
     uint16* pNwkAddrAssocDevList;
} ZPS_tsAplZdpIeeeAddrRsp;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpIeeeAddrRequest()**.
- u64IeeeAddrRemoteDevis the IEEE address of the remote node that sent the response (this is the IEEE address that was requested).
- u16NwkAddrRemoteDevis the network address of the remote node that sent the response (this is the network address specified in the original request).
- u8NumAssocDevis the number of neighboring nodes for which network addresses are also being reported (in the remainder of the structure).
- u8StartIndexis the index in the remote node's Neighbor table of the first entry to be included in this report. This element should be ignored if the element u8NumAssocDevis 0.
- pNwkAddrAssocDevListis a pointer to a list of 16-bit network addresses of the remote node's neighbors (this is a variable-length list with four bytes per node). This element should be ignored if the element u8NumAssocDevis 0.

Parent topic:ZDP response structures

ZPS_tsAplZdpNodeDescRsp This structure is used to store Node_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpNodeDescRequest()**. This response contains the Node descriptor of the node with a given network address.

The ZPS_tsAplZdpNodeDescRspstructure is detailed below.

```
typedef struct {
    uint8      u8Status;
    uint16     u16NwkAddrOfInterest;
    /* Rest of the message is variable length */
    ZPS_tsAplZdpNodeDescriptor tsNodeDescriptor;
} ZPS_tsAplZdpNodeDescRsp;
```

where:

- u8Status is the return status for **ZPS_eAplZdpNodeDescRequest()**.
- u16NwkAddrOfInterest is the network address of the remote node that sent the response (this is the network address that was specified in the request).
- tsNodeDescriptor is the returned Node descriptor, a structure of type ZPS_tsAplZdpNodeDescriptor (detailed in Section 9.2.1.1). This is only included if u8Status reports success.

Parent topic: ZDP response structures

ZPS_tsAplZdpPowerDescRsp This structure is used to store Power_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpPowerDescRequest()**. This response contains the Power descriptor of the node with a given network address.

The ZPS_tsAplZdpPowerDescRsp structure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint16 u16NwkAddrOfInterest;
    /* Rest of the message is variable length */
    ZPS_tsAplZdpNodePowerDescriptor sPowerDescriptor;
} ZPS_tsAplZdpPowerDescRsp;
```

where:

- u8Status is the return status for **ZPS_eAplZdpPowerDescRequest()**
- u16NwkAddrOfInterest is the network address of the remote node that sent the response (this is the network address that was specified in the request)
- sPowerDescriptor is the returned Power descriptor, a structure of type ZPS_tsAplZdpNodePowerDescriptor (detailed in Section 9.2.1.2). This is only included if u8Status reports success

Parent topic: ZDP response structures

ZPS_tsAplZdpSimpleDescRsp This structure is used to store Simple_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpSimpleDescRequest()**. This response contains the Simple descriptor of a given endpoint on the node with a given network address.

The ZPS_tsAplZdpSimpleDescRsp structure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint16 u16NwkAddrOfInterest;
    uint8 u8Length;
    /* Rest of the message is variable length */
    ZPS_tsAplZdpSimpleDescType sSimpleDescriptor;
} ZPS_tsAplZdpSimpleDescRsp;
```

where:

- u8Status is the return status for **ZPS_eAplZdpSimpleDescRequest()**.
- u16NwkAddrOfInterest is the network address of the remote node that sent the response (this is the network address that was specified in the request).
- u8Length is the length of the returned Simple descriptor, in bytes (depends on the number of clusters supported by the endpoint).
- sSimpleDescriptor is the returned Simple descriptor, a structure of type ZPS_tsAplZdpSimpleDescType (detailed in Section 9.2.1.3). This is only included if u8Status reports success.

Parent topic:ZDP response structures

ZPS_tsAplZdpExtendedSimpleDescRsp This structure is used to store Extended_Simple_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpExtendedSimpleDescRequest()**. This response contains a cluster list (combined input and output) for a given endpoint on the node with a given network address.

The ZPS_tsAplZdpExtendedSimpleDescRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint16 u16NwkAddr;
    uint8 u8EndPoint;
    uint8 u8AppInputClusterCount;
    uint8 u8AppOutputClusterCount;
    uint8 u8StartIndex;
    /* Rest of the message is variable length */
    uint16* pAppClusterList;
} ZPS_tsAplZdpExtendedSimpleDescRsp;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpExtendedSimpleDescRequest()**
- u16NwkAddris the network address of the remote node that sent the response (this is the network address that was specified in the request)
- u8EndPointis the number of the endpoint for which the response was sent (this is the endpoint number that was specified in the request)
- u8AppInputClusterCountis the total number of input clusters in the endpoint's complete input cluster list
- u8AppOutputClusterCountis the total number of output clusters in the endpoint's complete output cluster list
- u8StartIndexis the index, in the endpoint's complete input or output cluster list, of the first cluster reported in this response
- pAppClusterListis a pointer to the reported cluster list, input clusters first then output clusters. This is only included if u8Statusreports success

Parent topic:ZDP response structures

ZPS_tsAplZdpComplexDescRsp This structure is used to store Complex_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpComplexDescRequest()**. This response contains the Complex descriptor of the node with a given network address.

The ZPS_tsAplZdpComplexDescRspstructure is detailed below.

```
typedef struct {
        uint8 u8Status;
        uint16 u16NwkAddrOfInterest;
        uint8 u8Length;
        /* Rest of the message is variable Length */
        ZPS_tsAplZdpComplexDescElement sComplexDescriptor;
} ZPS_tsAplZdpComplexDescRsp;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpComplexDescRequest()**.
- u16NwkAddrOfInterestis the network address of the remote node that sent the response (this is the network address that was specified in the request).

- u8Lengthis the length of the returned Complex descriptor, in bytes.
- sComplexDescriptoris the returned Complex descriptor, a structure of type. ZPS_tsAplZdpComplexDescRsp(described below). This is only included if u8Statusreports success .

ZPS_tsAplZdpComplexDescElement

```
typedef struct { uint8 u8XMLTag;
        uint8 u8FieldCount;
        uint8 *pu8Data;
} ZPS_tsAplZdpComplexDescElement;
```

where:

- u8XMLTagis the XML tag for the current field.
- u8FieldCountis the number of fields in the Complex descriptor.
- *pu8Datais a pointer to the data of the current field.

Parent topic:ZPS_tsAplZdpComplexDescRsp

Parent topic:ZDP response structures

ZPS_tsAplZdpUserDescRsp This structure is used to store User_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpUserDescRequest()**. This response contains the User descriptor of the node with a given network address.

The ZPS_tsAplZdpUserDescRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint16 u16NwkAddrOfInterest;
    uint8 u8Length;
    /* Rest of the message is variable Length */
    char szUserDescriptor[ZPS_ZDP_LENGTH_OF_USER_DESC];
} ZPS_tsAplZdpUserDescRsp;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpUserDescRequest()**.
- u16NwkAddrOfInterestis the network address of the remote node that sent the response (this is the network address that was specified in the request).
- u8Lengthis the length of the returned User descriptor, in bytes (maximum: 16).
- szUserDescriptoris the returned User descriptor as a character array. This is only included if u8Statusreports success.

Parent topic:ZDP response structures

ZPS_tsAplZdpMatchDescRsp This structure is used to store Match_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpMatchDescRequest()**. This response contains details of the endpoints on the remote node that matched the criteria specified in the original request.

The ZPS_tsAplZdpMatchDescRspstructure is detailed below.

```
typedef struct {
        uint8 u8Status;
        uint16 u16NwkAddrOfInterest;
```

(continues on next page)

(continued from previous page)

```
    uint8 u8MatchLength;
    /* Rest of message is variable length */
    uint8* u8MatchList;
} ZPS_tsAplZdpMatchDescRsp;
```

where:

- u8Status is the return status for **ZPS_eAplZdpMatchDescRequest()**.
- u16NwkAddrOfInterest is the network address of the remote node that sent the response (this is the network address that was specified in the request).
- u8MatchLength is the length of the list of matched endpoints, in bytes.
- u8MatchList is a pointer to the list of matched endpoints, where each endpoint is represented by an 8-bit value (in the range 1-240).

Parent topic: ZDP response structures

ZPS_tsAplZdpActiveEpRsp This structure is used to store Active_EP_rsp message data - a response to a call to the function **ZPS_eAplZdpActiveEpRequest()**. This response contains a list of the active endpoints on a given network node.

The ZPS_tsAplZdpActiveEpRsp structure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint16 u16NwkAddrOfInterest;
    uint8 u8ActiveEpCount;
    /* Rest of the message is variable */
    uint8* pActiveEpList;
} ZPS_tsAplZdpActiveEpRsp;
```

where:

- u8Status is the return status for **ZPS_eAplZdpActiveEpRequest()**.
- u16NwkAddrOfInterest is the network address of the remote node that sent the response (this is the network address that was specified in the request).
- u8ActiveEpCount is the number of active endpoints on the node.
- pActiveEpList is a pointer to the list of active endpoints, where each endpoint is represented by an 8-bit value (in the range 1-240).

Parent topic: ZDP response structures

ZPS_tsAplZdpExtendedActiveEpRsp This structure is used to store Extended_Active_EP_rsp message data - a response to a call to the function **ZPS_eAplZdpExtendedActiveEpRequest()**. This response contains a list of the active endpoints on the node with a given network address.

The ZPS_tsAplZdpExtendedActiveEpRsp structure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint16 u16NwkAddr;
    uint8 u8ActiveEpCount;
    uint8 u8StartIndex;
    /* Rest of the message is variable Length */
    uint8* pActiveEpList;
} ZPS_tsAplZdpExtendedActiveEpRsp;
```

where:

- u8Status is the return status for **ZPS_eAplZdpExtendedActiveEpRequest()**.
- 16NwkAddr is the network address of the remote node that sent the response (this is the network address that was specified in the request).
- u8ActiveEpCount is the total number of active endpoints on the node.
- u8StartIndex is the index, in the node's list of active endpoints, of the first endpoint reported in this response.
- pActiveEpList is a pointer to the reported list of active endpoints (starting with the endpoint with index u8StartIndex).

Parent topic:ZDP response structures

ZPS_tsAplZdpUserDescConf This structure is used to store User_Desc_conf message data - a response to a call to the function **ZPS_eAplZdpUserDescSetRequest()**. This response contains a confirmation of the requested configuration of the User descriptor on a given network node.

The ZPS_tsAplZdpUserDescConf structure is detailed below.

```
typedef struct {
        uint8 u8Status;
        uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpUserDescConf;
```

where:

- u8Status is the return status for **ZPS_eAplZdpUserDescSetRequest()**.
- u16NwkAddrOfInterest is the network address of the remote node that sent the response (this is the network address that was specified in the request).

Parent topic:ZDP response structures

ZPS_tsAplZdpSystemServerDiscoveryRsp This structure is used to store System_Server_Discovery_rsp message data - a response to a call to the function **ZPS_eAplZdpSystemServerDiscoveryRequest()**. This response indicates which of the requested services are supported by a given network node.

The ZPS_tsAplZdpSystemServerDiscoveryRsp structure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint16 u16ServerMask;
} ZPS_tsAplZdpSystemServerDiscoveryRsp;
```

where:

- u8Status is the return status for the function **ZPS_eAplZdpSystemServerDiscoveryRequest()**.
- u16ServerMask is the returned bitmask that summarizes the requested services supported by the node (1 for 'supported', 0 for 'not supported' or 'not requested'). This bitmask is detailed in the table below.

Services Bitmask

Bit	Service
0	Primary Trust Centre
1	Backup Trust Centre
2	Primary Binding Table Cache
3	Backup Binding Table Cache
4	Primary Discovery Cache
5	Back-up Discovery Cache
6	Network Manager
7-15	Reserved

Parent topic:ZDP response structures

ZPS_tsAplZdpDiscoveryCacheRsp This structure is used to store Discovery_Cache_rsp message data - a response to a call to the function **ZPS_eAplZdpDiscoveryCacheRequest()**. This response indicates that the sending node has a primary discovery cache.

The ZPS_tsAplZdpDiscoveryCacheRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpDiscoveryCacheRsp;
```

where u8Statusis the return status for **ZPS_eAplZdpDiscoveryCacheRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpDiscoveryStoreRsp This structure is used to store Discovery_Store_rsp message data - a response to a call to the function **ZPS_eAplZdpDiscoveryStoreRequest()**. This response indicates whether the sending node has successfully reserved space in its primary discovery cache.

The ZPS_tsAplZdpDiscoveryStoreRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpDiscoveryStoreRsp;
```

where u8Statusis the return status for **ZPS_eAplZdpDiscoveryStoreRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpNodeDescStoreRsp This structure is used to store Node_Desc_store_rsp message data - a response to a call to the function **ZPS_eAplZdpNodeDescStoreRequest()**. This response indicates whether the sending node has successfully stored the received Node descriptor in its primary discovery cache.

The ZPS_tsAplZdpNodeDescStoreRspstructure is detailed below.

```
typedef struct {
     uint8 u8Status;
} ZPS_tsAplZdpNodeDescStoreRsp;
```

where u8Statusis the return status for **ZPS_eAplZdpNodeDescStoreRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpPowerDescStoreRsp This structure is used to store Power_Desc_store_rsp message data - a response to a call to the function **ZPS_eAplZdpPowerDescStoreRequest()**. This response indicates whether the sending node has successfully stored the received Power descriptor in its primary discovery cache.

The ZPS_tsAplZdpPowerDescStoreRspstructure is detailed below.

```
typedef struct {
        uint8 u8Status;
        uint64 u64IeeeAddr;
        /* Rest of message is variable length */
        ZPS_tsAplZdpNodePowerDescriptor sPowerDescriptor;
} ZPS_tsAplZdpPowerDescStoreRsp;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpPowerDescStoreRequest()**.
- u64IeeeAddris the IEEE/MAC address of the device whose Power descriptor has been stored in the primary discovery cache.
- sPowerDescriptoris the Power descriptor stored (see Section 9.2.1.1).

Parent topic:ZDP response structures

ZPS_tsAplZdpSimpleDescStoreRsp This structure is used to store Power_Desc_store_rsp message data - a response to a call to the function **ZPS_eAplZdpSimpleDescStoreRequest()**. This response indicates whether the sending node has successfully stored the received Simple descriptor in its primary discovery cache.

The ZPS_tsAplZdpSimpleDescStoreRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpSimpleDescStoreRsp;
```

where u8Statusis the return status for **ZPS_eAplZdpSimpleDescStoreRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpActiveEpStoreRsp This structure is used to store Active_EP_store_rsp message data - a response to a call to the function **ZPS_eAplZdpActiveEpStoreRequest()**. This response indicates whether the sending node has successfully stored the received list of active endpoints in its primary discovery cache.

The ZPS_tsAplZdpActiveEpStoreRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpActiveEpStoreRsp;
```

where u8Statusis the return status for **ZPS_eAplZdpActiveEpStoreRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpFindNodeCacheRsp This structure is used to store Find_node_cache_rsp message data - a response to a call to the function **ZPS_eAplZdpFindNodeCacheRequest()**. This response indicates that the sending node holds 'discovery information' about a given network node in its primary discovery cache.

The ZPS_tsAplZdpFindNodeCacheRspstructure is detailed below.

```
typedef struct {
    uint16 u16CacheNwkAddr;
    uint16 u16NwkAddr;
    uint64 u64IeeeAddr;
} ZPS_tsAplZdpFindNodeCacheRsp;
```

where:

- u16CacheNwkAddris the network address of the remote node that sent the response.
- u16NwkAddris the network address of the node of interest (this is the network address that was specified in the request).
- u64IeeeAddris the IEEE address of the node of interest (this is the IEEE address that was specified in the request).

Parent topic:ZDP response structures

ZPS_tsAplZdpRemoveNodeCacheRsp This structure is used to store Remove_node_cache_rsp message data - a response to a call to the function **ZPS_eAplZdpRemoveNodeCacheRequest()**. This response indicates whether the sending node has successfully removed from its primary discovery cache all 'discovery information' relating to a given End Device node.

The ZPS_tsAplZdpRemoveNodeCacheRspstructure is detailed below.

```
typedef struct {
     uint8 u8Status;
} ZPS_tsAplZdpRemoveNodeCacheRsp;
```

where u8Statusis the return status for the function **ZPS_eAplZdpRemoveNodeCacheRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpEndDeviceBindRsp This structure is used to store End_Device_Bind_rsp message data - a response to a call to the function **ZPS_eAplZdpEndDeviceBindRequest()**. This response is issued by the Coordinator to indicate the status of an End Device binding request.

The ZPS_tsAplZdpEndDeviceBindRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpEndDeviceBindRsp;
```

where u8Statusis the return status for **ZPS_eAplZdpEndDeviceBindRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpBindRsp This structure is used to store Bind_rsp message data - a response to a call to the function **ZPS_eAplZdpBindUnbindRequest()**. This response indicates the status of a binding request (a request to modify of a binding table).

The ZPS_tsAplZdpBindRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpBindRsp;
```

where u8Statusis the return status for **ZPS_eAplZdpBindUnbindRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpUnbindRsp This structure is used to store Unbind_rsp message data - a response to a call to the function **ZPS_eAplZdpBindUnbindRequest()**. This response indicates the status of an unbinding request (a request to modify of a binding table).

The ZPS_tsAplZdpUnbindRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpUnbindRsp;
```

where u8Statusis the return status for **ZPS_eAplZdpBindUnbindRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpBindRegisterRsp This structure is used to store Bind_Register_rsp message data - a response to a call to the function **ZPS_eAplZdpBindRegisterRequest()**. This response contains binding information held on the responding node concerning the requesting node.

The ZPS_tsAplZdpBindRegisterRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint16 u16BindingTableEntries;
    uint16 u16BindingTableListCount;
    /* Rest of the message is variable Length */
    ZPS_tsAplZdpBindingTable sBindingTableList;
} ZPS_tsAplZdpBindRegisterRsp;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpBindRegisterRequest()**.
- u16BindingTableEntriesis the total number of binding table entries concerning the requesting node held on the responding node.
- u16BindingTableListCountis the number of binding table entries concerning the requesting node contained in this response.
- sBindingTableListis a pointer to the first item in the list of reported binding table entries. A list item is of type ZPS_tsAplZdpBindingTabledetailed below.

ZPS_tsAplZdpBindingTable

```
typedef struct
{
uint64 u64SourceAddress;
ZPS_tsAplZdpBindingTableEntry* psBindingTableEntryForSpSrcAddr;
}ZPS_tsAplZdpBindingTable;
```

where:

- u64SourceAddressis the IEEE address of the node to which the binding table entry relates.
- psBindingTableEntryForSpSrcAddris a pointer to the relevant binding table information. This information is contained in a structure of type ZPS_tsAplZdpBindingTableEntrydetailed below.

Parent topic:ZPS_tsAplZdpBindRegisterRsp

ZPS_tsAplZdpBindingTableEntry

```
typedef struct
{
    uint8 u8SourceEndpoint;
    uint16 u16ClusterId;
    uint8 u8DstAddrMode;
    union {
        struct {
        uint16 u16DstAddress;
        } sShort;
        struct {
        uint64 u64DstAddress;
        uint8 u8DstEndPoint;
        } sExtended;
    };
}ZPS_tsAplZdpBindingTableEntry;
```

where:

- u8SourceEndpointis the number of the bound endpoint (1-240) on the source node of the binding
- u16ClusterIdis the ID of the cluster involved in the binding, on the source node of the binding
- u8DstAddrModeis the addressing mode used in the rest of the structure (see Table 19 below)
- u16DstAddressis the network address of the destination node of the binding (this is only application if u8DstAddrModeis set to 0x03)
- u64DstAddressis the IEEE address of the destination node of the binding (this is only application if u8DstAddrModeis set to 0x04)
- u8DstEndPointis the number of the bound endpoint (1-240) on the destination node of the binding

Addressing modes

u8DstAddrMode	Code	Description
0x00	ZPS_E_ADDR_MODE_BOUND	Bound endpoint
0x01	ZPS_E_ADDR_MODE_GROUP	16-bit Group address
0x02	ZPS_E_ADDR_MODE_SHORT	16-bit Network (Short) address
0x03	ZPS_E_ADDR_MODE_IEEE	64-bit IEEE/MAC address

Parent topic:ZPS_tsAplZdpBindRegisterRsp

Parent topic:ZDP response structures

ZPS_tsAplZdpReplaceDeviceRsp This structure is used to store Replace_Device_rsp message data - a response to a call to the function **ZPS_eAplZdpReplaceDeviceRequest()**. This response indicates the status of the replace request.

The ZPS_tsAplZdpReplaceDeviceRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpReplaceDeviceRsp;
```

where u8Statusis the return status for **ZPS_eAplZdpReplaceDeviceRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpStoreBkupBindEntryRsp This structure is used to store Store_Bkup_Bind_Entry_rsp message data - a response to a call to the function **ZPS_eAplZdpStoreBkupBindEntryRequest()**. This response indicates the status of the back-up request.

The ZPS_tsAplZdpStoreBkupBindEntryRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpStoreBkupBindEntryRsp;
```

where u8Statusis the return status for the function **ZPS_eAplZdpStoreBkupBindEntryRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpRemoveBkupBindEntryRsp This structure is used to store Remove_Bkup_Bind_Entry_rsp message data - a response to a call to the function **ZPS_eAplZdpRemoveBkupBindEntryRequest()**. This response indicates the status of the remove request.

The ZPS_tsAplZdpRemoveBkupBindEntryRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpRemoveBkupBindEntryRsp;
```

where u8Statusis the return status for the function **ZPS_eAplZdpRemoveBkupBindEntryRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpBackupBindTableRsp This structure is used to store Backup_Bind_Table_rsp message data - a response to a call to the function **ZPS_eAplZdpBackupBindTableRequest()**. This response indicates the status of the back-up request.

The ZPS_tsAplZdpBackupBindTableRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint16 u16EntryCount;
} ZPS_tsAplZdpBackupBindTableRsp;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpBackupBindTableRequest()**
- u16EntryCountis the number of binding table entries that have been backed up

Parent topic:ZDP response structures

ZPS_tsAplZdpRecoverBindTableRsp This structure is used to store Recover_Bind_Table_rsp message data - a response to a call to the function **ZPS_eAplZdpRecoverBindTableRequest()**. This response indicates the status of the recover request and contains the recovered binding table entries.

The ZPS_tsAplZdpRecoverBindTableRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint16 u16StartIndex;
    uint16 u16BindingTableEntries;
    uint16 u16BindingTableListCount;
```

(continues on next page)

(continued from previous page)

```
    /* Rest of the message is variable length */
    ZPS_tsAplZdpBindingTable sBindingTableList;
} ZPS_tsAplZdpRecoverBindTableRsp;
```

where:

- u8Status is the return status for **ZPS_eAplZdpRecoverBindTableRequest()**
- u16StartIndex is the binding table index of the first entry in the set of recovered binding table entries (sBindingTableList)
- u16BindingTableEntries is the total number of entries in the back-up binding table cache
- u16BindingTableListCount is the number of entries in the set of recovered binding table entries (sBindingTableList)
- sBindingTableList is a pointer to the first item in the list of recovered binding table entries. A list item is of type ZPS_tsAplZdpBindingTable, detailed in Section 8.2.3.26.

Parent topic: ZDP response structures

ZPS_tsAplZdpBackupSourceBindRsp This structure is used to store Backup_Source_Bind_rsp message data - a response to a call to the function **ZPS_eAplZdpBackupSourceBindRequest()**. This response indicates the status of the back-up request.

The ZPS_tsAplZdpBackupSourceBindRsp structure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpBackupSourceBindRsp;
```

where u8Status is the return status for the function **ZPS_eAplZdpBackupSourceBindRequest()**.

Parent topic: ZDP response structures

ZPS_tsAplZdpRecoverSourceBindRsp This structure is used to store Recover_Source_Bind_rsp message data - a response to a call to the function **ZPS_eAplZdpRecoverSourceBindRequest()**. This response indicates the status of the recover request and contains the recovered binding table entries.

The ZPS_tsAplZdpRecoverSourceBindRsp structure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint16 u16StartIndex;
    uint16 u16SourceTableEntries;
    uint16 u16SourceTableListCount;
    /* Rest of the message is variable length */
    uint64* pu64SourceTableList;
} ZPS_tsAplZdpRecoverSourceBindRsp;
```

where:

- u8Status is the return status for the function **ZPS_eAplZdpRecoverSourceBindRequest()**.
- u16StartIndex is the binding table index of the first entry in the set of recovered binding table entries (pu64SourceTableList).
- u16SourceTableEntries is the total number of source binding table entries in the back-up binding table cache.
- u16SourceTableListCount is the number of entries in the set of recovered binding table entries. (pu64SourceTableList).

- pu64SourceTableListis a pointer to the first item in the list of recovered binding table entries.

Parent topic:ZDP response structures

ZPS_tsAplZdpMgmtNwkDiscRsp This structure is used to store Mgmt_NWK_Disc_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtNwkDiscRequest()**. This response reports the networks discovered in a network discovery (all the networks or a subset).

The ZPS_tsAplZdpMgmtNwkDiscRspstructure is detailed below.

```
typedef struct {
  uint8 u8Status;
  uint8 u8NetworkCount;
  uint8 u8StartIndex;
  uint8 u8NetworkListCount;
  /* Rest of the message is variable length */
  ZPS_tsAplZdpNetworkDescr* psNetworkDescrList;
} ZPS_tsAplZdpMgmtNwkDiscRsp;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpMgmtNwkDiscRequest()**
- u8NetworkCountis the total number of networks discovered
- u8StartIndexis the index, in the complete list of discovered networks, of the first network reported in this response (through psNetworkDescrList)
- u8NetworkListCountis the number of discovered networks reported in this response (through psNetworkDescrList)
- psNetworkDescrListis a pointer to the first entry in a list of network descriptors for the discovered networks. Each entry is of the type ZPS_tsAplZdpNetworkDescrdetailed below.

  ZPS_tsAplZdpNetworkDescr

  ```
  typedef struct
  {
      uint64 u64ExtPanId;
      uint8 u8LogicalChan;
      uint8 u8StackProfile;
      uint8 u8ZigBeeVersion;
      uint8 u8PermitJoining;
      uint8 u8RouterCapacity;
      uint8 u8EndDeviceCapacity;
  } ZPS_tsAplZdpNetworkDescr;
  ```

  where:

  - u64ExtPanIdis the 64-bit extended PAN ID of the discovered network.
  - u8LogicalChanis the radio channel in which the discovered network operates (value in range 0 to 26, but only channels 11 to 26 relevant to 2400-MHz band).
  - u8StackProfileis the 4-bit identifier of the ZigBee stack profile used by the discovered network (0 - manufacturer-specific, 1 - ZigBee, 2 - ZigBee PRO, other values reserved) and is fixed at 2 for the NXP stack.
  - u8ZigBeeVersionis the 4-bit version of the ZigBee protocol used by the discovered network.
  - u8PermitJoiningindicates whether the discovered network is currently allowing joinings - that is, at least one node (a Router or the Coordinator) of the network is allowing other nodes to join it:

    * 0x01: Joinings allowed.
    * 0x00: Joinings not allowed.
    * All other values reserved.
  - u8RouterCapacityindicates whether the device is capable of accepting join requests from Routers - set to TRUE if capable, FALSE otherwise.
  - u8EndDeviceCapacityindicates whether the device is capable of accepting join requests from End Devices - set to TRUE capable, FALSE otherwise.

Parent topic:ZDP response structures

ZPS_tsAplZdpMgmtLqiRsp This structure is used to store Mgmt_Lqi_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtLqiRequest()**. This response reports a list of neighboring nodes along with their LQI (link quality) values.

The ZPS_tsAplZdpMgmtLqiRspstructure is detailed below.

```
typedef struct {
  uint8 u8Status;
  uint8 u8NeighborTableEntries;
  uint8 u8StartIndex;
  uint8 u8NeighborTableListCount;
  /* Rest of the message is variable length */
  ZPS_tsAplZdpNtListEntry* pNetworkTableList;
} ZPS_tsAplZdpMgmtLqiRsp;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpMgmtLqiRequest()**
- u8NeighborTableEntriesis the total number of Neighbor table entries on the remote node
- u8StartIndexis the Neighbor table index of the first entry reported in this response (through pNetworkTableList)
- u8NetworkListCountis the number of Neighbor table entries reported in this response (through pNetworkTableList)
- pNetworkTableListis a pointer to the first entry in the list of reported Neighbor table entries. Each entry is of the type ZPS_tsAplZdpNtListEntrydetailed below.

  ZPS_tsAplZdpNtListEntry

```
typedef struct
{
    uint64 u64ExtPanId;
    uint64 u64ExtendedAddress;
    uint16 u16NwkAddr;
    uint8 u8LinkQuality;
    uint8 u8Depth;
    /*
    * Bitfields are used for syntactic neatness and space saving.
    * May need to assess whether these are suitable for embedded
    environment and may need to watch endianness on u8Assignment
    */
  union
    {
      struct
      {
        unsigned u1Reserved1:1;
        unsigned u2Relationship:3;
        unsigned u2RxOnWhenIdle:2;
```

(continues on next page)

(continued from previous page)

```
        unsigned u2DeviceType:2;
        unsigned u6Reserved2:6;
        unsigned u2PermitJoining:2;
      } ;
      uint8 au8Field[2];
  } uAncAttrs;
} ZPS_tsAplZdpNtListEntry;
```

where:

- u64ExtPanIdis the 64-bit extended PAN ID of the network .
- u64ExtendedAddressis the IEEE address of the neighboring node.
- u16NwkAddris the network address of the neighboring node.
- u8LinkQualityis the estimated LQI (link quality) value for radio transmissions from the neighboring node.
- u8Depthis the tree depth of the neighboring node (where the Coordinator is at depth zero).
- u1Reserved1:1is a 1-bit reserved value and should be set zero.
- u2Relationship:3is a 3-bit value representing the neighboring node's relationship to the local node:
  - 0: Neighbor is the parent.
  - 1: Neighbor is a child.
  - 2: Neighbor is a sibling (has same parent).
  - 3: None of the above.
  - 4: Neighbor is a former child.
- u2RxOnWhenIdle:2is a 2-bit value indicating whether the neighboring node's receiver is enable during idle periods:
  - 0: Receiver off when idle (sleeping device)
  - 1: Receiver on when idle (non-sleeping device)
  - 2: Unknown
- u2DeviceType:2is a 2-bit value representing the ZigBee device type of the neighboring node:
  - 0: Coordinator
  - 1: Router
  - 2: End Device
  - 3: Unknown
- u6Reserved2:6is a 6-bit reserved value and should be set zero.
- u2PermitJoining:2is a 2-bit value indicating whether the neighboring node is accepting joining requests:
  - 0: Not accepting join requests
  - 1: Accepting join requests
  - 2: Unknown
- au8Field[2]is the allocation of two bytes for the union.

Parent topic:ZDP response structures

ZPS_tsAplZdpMgmtRtgRsp This structure is used to store Mgmt_Rtg_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtRtgRequest()**. This response reports the contents of the remote node's Routing table

The ZPS_tsAplZdpMgmtRtgRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint8 u8RoutingTableEntries;
    uint8 u8StartIndex;
    uint8 u8RoutingTableCount;
    /* Rest of the message is variable length */
    ZPS_tsAplZdpRtEntry* pRoutingTableList;
} ZPS_tsAplZdpMgmtRtgRsp;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpMgmtRtgRequest()**
- u8RoutingTableEntriesis the total number of Routing table entries on the remote node
- u8StartIndexis the Routing table index of the first entry reported in this response (through pRoutingTableList)
- u8RoutingTableCountis the number of Routing table entries reported in this response (through pRoutingTableList)
- pRoutingTableListis a pointer to the first entry in the list of reported Routing table entries. Each entry is of the type ZPS_tsAplZdpRtEntrydetailed below

```
typedef struct
{
    uint16 u16NwkDstAddr; /**< Destination Network address */
    uint16 u16NwkNxtHopAddr; /**< Next hop Network address */
  union
    {
    struct
       {
        unsigned u3Status:3;
        unsigned u1MemConst:1;
        unsigned u1ManyToOne:1;
        unsigned u1RouteRecordReqd:1;
        unsigned u1Reserved:2;
        } bfBitfields;
        uint8 u8Field;
  } uAncAttrs;
} ZPS_tsAplZdpRtEntry;
```

where:

  - u16NwkDstAddris the destination network address of the route.
  - u16NwkNxtHopAddris the 'next hop' network address of the route.
  - u3Status:3is the 3-bit status for the route:
    - 000 = ACTIVE
    - 001 = DISCOVERY_UNDERWAY
    - 010 = DISCOVERY_FAILED
    - 011 = INACTIVE
    - 100 = VALIDATION_UNDERWAY
    - 101-111 = Reserved.

- u1MemConst:1is a bit indicating whether the device is a memory-constrained concentrator.
- u1ManyToOne:1is a bit indicating whether the destination node is a concentrator that issued a many-to-one request.
- u1RouteRecordReqd:1is a bit indicating whether a route record command frame. should be sent to the destination before the next data packet.
- u1Reserved:2are reserved bits.
- u8Fieldcontains the full set of flags of the bfBitfieldssub-structure, with u3Status:3occupying the most significant bits and u1Reserved:2occupying the least significant bits (for a big-endian device).

Parent topic:ZDP response structures

ZPS_tsAplZdpMgmtBindRsp This structure is used to store Mgmt_Bind_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtBindRequest()**. This response reports the contents of the remote node's Binding table.

The ZPS_tsAplZdpMgmtBindRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint16 u16BindingTableEntries;
    uint16 u16StartIndex;
    uint16 u16BindingTableListCount;
    /* Rest of the message is variable length */
    ZPS_tsAplZdpBindingTable sBindingTableList;
} ZPS_tsAplZdpMgmtBindRsp;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpMgmtBindRequest()**
- u16BindingTableEntriesis the total number of Binding table entries on the remote node
- u8StartIndexis the Binding table index of the first entry reported in this response (through sBindingTableList)
- u16BindingTableListCountis the number of Binding table entries reported in this response (through sBindingTableList)
- sBindingTableListis a pointer to the first entry in the list of reported Binding table entries. Each entry is of the type ZPS_tsAplZdpBindingTable, detailed in Section 9.2.2.29

Parent topic:ZDP response structures

ZPS_tsAplZdpMgmtLeaveRsp This structure is used to store Mgmt_Leave_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtLeaveRequest()**. This response is issued by a remote node that has been requested to leave the network.

The ZPS_tsAplZdpMgmtLeaveRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpMgmtLeaveRsp;
```

where u8Statusis the return status for **ZPS_eAplZdpMgmtLeaveRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpMgmtDirectJoinRsp This structure is used to store Mgmt_Direct_Join_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtDirectJoinRequest()**. This response is issued by a remote node (Router or Coordinator) that has been requested to allow a particular device to join the network as a child of the node.

The ZPS_tsAplZdpMgmtDirectJoinRspstructure is detailed below.

```
typedef struct {
        uint8 u8Status;
} ZPS_tsAplZdpMgmtDirectJoinRsp;
```

where u8Statusis the return status for **ZPS_eAplZdpMgmtDirectJoinRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpMgmtPermitJoiningRsp This structure is used to store Mgmt_Permit_Joining_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtPermitJoiningRequest()**. This response is issued by a remote node (Router or Coordinator) that has been requested to enable or disable joining for a specified amount of time. The response is only sent if the original request was unicast (and not if it was broadcast).

The ZPS_tsAplZdpMgmtPermitJoiningRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpMgmtPermitJoiningRsp;
```

where u8Statusis the return status for the function **ZPS_eAplZdpMgmtPermitJoiningRequest()**.

Parent topic:ZDP response structures

ZPS_tsAplZdpMgmtCacheRsp This structure is used to store Mgmt_Cache_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtCacheRequest()**. This response reports a list of the End Devices registered in the node's primary discovery cache.

The ZPS_tsAplZdpMgmtCacheRspstructure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint8 u8DiscoveryCacheEntries;
    uint8 u8StartIndex;
    uint8 u8DiscoveryCacheListCount;
   /* Rest of the message is variable length */
   ZPS_tsAplDiscoveryCache* pDiscoveryCacheList;
} ZPS_tsAplZdpMgmtCacheRsp;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpMgmtCacheRequest()**
- u8DiscoveryCacheEntriesis the total number of discovery cache entries on the remote node.
- u8StartIndexis the discovery cache index of the first entry reported in this response (through pDiscoveryCacheList).
- u8DiscoveryCacheListCountis the number of discovery cache entries reported in this response (through pDiscoveryCacheList).
- pRoutingTableListis a pointer to the first entry in the list of reported discovery cache entries. Each entry is of the type ZPS_tsAplDiscoveryCachedetailed below.

ZPS_tsAplDiscoveryCache

```
typedef struct {
    uint64 u64ExtendedAddress;
    uint16 u16NwkAddress;
} ZPS_tsAplDiscoveryCache;
```

where:

- u64ExtendedAddressis the IEEE address of the End Device.
- u16NwkAddressis the network address of the End Device.

Parent topic:ZPS_tsAplZdpMgmtCacheRsp

Parent topic:ZDP response structures

ZPS_tsAplZdpMgmtNwkUpdateNotify This structure is used to store Mgmt_NWK_Update_notify message data - a notification which can be sent in response to a call to the function **ZPS_eAplZdpMgmtNwkUpdateRequest()**. This notification reports the results of an energy scan on the wireless network radio channels.

The ZPS_tsAplZdpMgmtNwkUpdateNotifystructure is detailed below.

Sample Codeblock:

```
typedef struct {
    uint8 u8Status;
    uint32 u32ScannedChannels;
    uint16 u16TotalTransmissions;
   uint16 u16TransmissionFailures;
   uint8 u8ScannedChannelListCount;
   /* Rest of the message is variable Length */
   uint8* u8EnergyValuesList;
} ZPS_tsAplZdpMgmtNwkUpdateNotify;
```

where:

- u8Statusis the return status for **ZPS_eAplZdpMgmtNwkUpdateRequest()**
- u32ScannedChannelsis a bitmask of the set of scanned radio channels (‘1’ means scanned, ‘0’ means not scanned):
  - Bits 0 to 26 respectively represent channels 0 to 26 (only bits 11 to 26 are relevant to the 2400-MHz band)
  - Bits 27 to 31 are reserved
- u16TotalTransmissionsis the total number of transmissions (from other networks) detected during the scan
- u16TransmissionFailuresis the number of failed transmissions detected during the scan
- u8ScannedChannelListCountis the number of energy-level measurements (one per scanned channel) reported in this notification (through u8EnergyValuesList)
- u8EnergyValuesListis a pointer to the first in the set of reported energy-level measurements (the value 0xFF indicates there is too much interference on the channel)

Parent topic:ZDP response structures

ZPS_tsAplZdpParentAnnceRsp This structure is used to store the data for a response to a Parent Announcement message that was sent using the function **ZPS_eAplZdpParentAnnceReq()**. This response reports any child nodes of the responding node that conflict with child nodes specified in the received Parent Announcement message.

The ZPS_tsAplZdpParentAnnceRspstructure is detailed below.

```
typedef struct {
   uint64* pu64ChildList;
   uint8 u8NumberOfChildren;
   uint8 u8Status;
} ZPS_tsAplZdpParentAnnceRsp;
```

where:

- pu64ChildListis a pointer to a list of 64-bit IEEE/MAC addresses of the child nodes in common.
- u8NumberOfChildrenis the number of child nodes in common.
- u8Statusis the status of the response.

Parent topic:ZDP response structures

Parent topic:ZDP structures

Parent topic:*ZigBee Device Profile (ZDP) API*

Broadcast addresses When sending a request using a ZDP API function, the request can be broadcast to all nodes in the network by specifying a special 16-bit network address (0xFFFF) or 64-bit IEEE/MAC address (0xFFFFFFFFFFFFFFFF). Other broadcast options are also available in order to target particular groups of nodes, as indicated in the table below.

Address Type	Broadcast Address	Target Nodes
Network (16-bit)	0xFFFF	All nodes in the network
0xFFFD	All nodes for which 'Rx on when idle' is TRUE	
0xFFFC	All Routers and the Coordinator	
IEEE/MAC (64-bit)	0xFFFFFFFFFFFFFFFF	All nodes in the network

Parent topic:*ZigBee Device Profile (ZDP) API*

General ZPS Resources The chapter describes the general API resources provided in the ZigBee PRO Stack (ZPS) software.

In this chapter:

- The **ZigBee Queue** resources are detailed in Section 10.1.
- The **ZigBee Timer** resources are detailed in Section 10.2.
- The **Critical Section and Mutex** resources are detailed in Section 10.3.

Note: Amongst the general resources, functions are supplied to allow the extraction of payload data from received ZDP packets. These resources are not documented here but are provided in the header file **appZdpExtractions.h**.

ZigBee Queue Resources The ZigBee Queue resources are concerned with creating and operating queues for passing messages from one task to another. These resources are provided in the header file **ZQueue.h**.

- The ZigBee Queue functions are described in Section 10.1.1.
- The ZigBee Queue structures are described in Section 10.1.2.

ZigBee queue functions The ZigBee Queue functions are listed below.

Function page

1. ZQ_vQueueCreate
2. ZQ_bQueueSend
3. ZQ_bQueueReceive
4. ZQ_bQueueIsEmpty
5. ZQ_u32QueueGetQueueSize
6. ZQ_u32QueueGetQueueMessageWaiting

Parent topic: ZigBee queue functions

ZQ_vQueueCreate

```
void ZQ_vQueueCreate(tszQueue *psQueueHandle,
                     const uint32 uiQueueLength,
                     const uint32 uiItemSize,
                     uint8 *pu8StartQueue);
```

Description

This function creates a message queue for use by the application or stack (message queues are described in Section 6.9.1). The size of the queue and the size of a message in the queue must be specified, as well as the location in memory where the queue should start. A unique handle must also be given to the queue, where this handle is a pointer to a tszQueuestructure that contains up-to-date information about the queue.

Parameters

- *psQueueHandle* Handle of message queue - this is a pointer to a tszQueue structure (see Section 10.1.2.1).
- *uiQueueLength* Size of the queue in terms of the number of messages that it can hold.
- *uiItemSize* Size of a message in the queue, in bytes.
- *pu8StartQueue* Pointer to the start of the message queue.

Parent topic:ZQ_vQueueCreate

Returns None

Parent topic:ZQ_vQueueCreate

Parent topic: ZigBee queue functions

ZQ_bQueueSend

```
bool_t ZQ_bQueueSend(void *pvQueueHandle,
                     const void *pvItemToQueue);
```

Description This function submits a message to the specified message queue. The return code indicates whether the message was successfully added to the queue.

Parent topic:ZQ_bQueueSend

Parameters

- *pvQueueHandle* Handle of message queue
- *pvItemToQueue* Pointer to the message to be added to the queue

Parent topic:ZQ_bQueueSend

Returns Boolean indicating the outcome of the operation:

- TRUE - message successfully added to the queue
- FALSE - message not added to the queue

Parent topic:ZQ_bQueueSend

Parent topic: ZigBee queue functions

ZQ_bQueueReceive

```
ZQ_bQueueReceive(void *pvQueueHandle,
                 void *pvItemFromQueue);
```

Description This function obtains a message from the specified message queue. The return code indicates whether a message was successfully obtained from the queue.

Parent topic:ZQ_bQueueReceive

Parameters

- *pvQueueHandle:* Handle of message queue
- *pvItemFromQueue:* Pointer to memory location to receive the obtained message

Parent topic:ZQ_bQueueReceive

Returns Boolean indicating the outcome of the operation:

- TRUE - message successfully obtained from the queue.
- FALSE - message not obtained from the queue.

Parent topic:ZQ_bQueueReceive

Parent topic: ZigBee queue functions

ZQ_bQueueIsEmpty

```
bool_t ZQ_bQueueIsEmpty(void *pvQueueHandle);
```

Description This function checks whether the specified message queue is empty. The return code indicates whether the queue is empty.

Parent topic:ZQ_bQueueIsEmpty

Parameters *pvQueueHandle* Handle of message queue

Parent topic:ZQ_bQueueIsEmpty

Returns Boolean indicating the outcome of the operation:

- TRUE - message queue is empty.
- FALSE - message queue is not empty.

Parent topic:ZQ_bQueueIsEmpty

Parent topic: ZigBee queue functions

ZQ_u32QueueGetQueueSize

```
bool_t ZQ_bQueueIsEmpty(void *pvQueueHandle);
```

Description This function obtains the capacity of the specified message queue. The return code indicates the size of the queue in terms of the number of messages that it can hold.

Parent topic:ZQ_u32QueueGetQueueSize

Parameters *pvQueueHandle* Handle of message queue

Parent topic:ZQ_u32QueueGetQueueSize

Returns The capacity of the queue in terms of the number of messages that it can hold.

Parent topic:ZQ_u32QueueGetQueueSize

Parent topic: ZigBee queue functions

ZQ_u32QueueGetQueueMessageWaiting

```
uint32 ZQ_u32QueueGetQueueMessageWaiting(
                void *pu8QueueHandle);
```

Description This function obtains the number of messages that are currently waiting in the specified message queue.

Parent topic:ZQ_u32QueueGetQueueMessageWaiting

Parameters *pvQueueHandle* Handle of message queue

Parent topic:ZQ_u32QueueGetQueueMessageWaiting

Returns Number of messages waiting in the queue

Parent topic:ZQ_u32QueueGetQueueMessageWaiting

Parent topic: ZigBee queue functions

Parent topic: ZigBee Queue Resources

ZigBee queue structures

tszQueue The ZigBee queue structure `tszQueue`is shown below.

```
typedef struct
{
uint32 u32Length;
uint32 u32ItemSize;
uint32 u32MessageWaiting;
void *pvHead;
void *pvWriteTo;
void *pvReadFrom;
}tszQueue;
```

where:

- u32Lengthis the size of the queue in terms of the number of messages that it can hold
- u32ItemSizeis the size of a message, in bytes
- u32MessageWaitingis the number of messages currently in the queue
- pvHeadis a pointer to the beginning of the queue storage area
- pvWriteTois a pointer to the next free place in the storage area where a new message can be written
- pvReadFromis a pointer to the next message to be read from the storage area

Parent topic: ZigBee queue structures

Parent topic: ZigBee Queue Resources

Parent topic:*General ZPS Resources*

ZigBee Timer resources The ZigBee Timer functions are concerned with initializing and operating software timers. These resources are provided in the header file **ZTimer.h**.

- The ZigBee Timer functions are described in Section 10.2.1
- The ZigBee Timer structures are described in Section 10.2.2

ZigBee Timer functions The functions are listed below.

Function page

1. ZTIMER_eInit
2. ZTIMER_eOpen
3. ZTIMER_eClose
4. ZTIMER_eStart
5. ZTIMER_eStop
6. ZTIMER_eGetState

To use the software timers, the while loop of your application must include a call to the following function:

```
void ZTIMER\_vTask\(void\);
```

Parent topic: ZigBee Timer functions

ZTIMER_eInit

```
ZTIMER_teStatus ZTIMER_eInit(ZTIMER_tsTimer *psTimers,
                             uint8 u8NumTimers);
```

Description This function initializes a set of software timers for use by the application. A list of timers is provided in an array, in which each array element is a structure containing information on one timer (see Section 10.2.2.1). The index of an array element is used as a reference for the corresponding timer.

In order to use one of the initialized timers, it must first be opened using **ZTIMER_eOpen()**.

Parent topic:ZTIMER_eInit

Parameters

- *psTimers:* Pointer to an array of structures, where each array element contains information for one timer (see Section 10.2.2.1)
- *u8NumTimers:* Number of timers in the above array

Parent topic:ZTIMER_eInit

Returns

- E_ZTIMER_OK (timers successfully initialized)
- E_ZTIMER_FAIL (timers not initialized)

Parent topic:ZTIMER_eInit

Parent topic: ZigBee Timer functions

ZTIMER_eOpen

```
ZTIMER_teStatus ZTIMER_eOpen(
                uint8 *pu8TimerIndex,
                ZTIMER_tpfCallback pfCallback,
                void *pvParams,
                uint8 u8Flags);
```

Description This function is used to open the specified software timer. A list of parameter values for the timer must be provided as well as a user-defined callback function that will be used to perform any operations required on expiration of the timer.

The callback function has the following prototype:

typedef void (*ZTIMER_tpfCallback)(void * *pvParam* **);**

where *pvParam* is a pointer to the timer parameters.

The function also includes a parameter *u8Flags*, which specifies whether the timer should allow or prevent sleep. When activity checks are made to decide whether the device can enter sleep mode, the value of this flag determines if the (running) timer will stop the device from going to sleep.

Before a timer is opened, it must have been initialized in a call to **ZTIMER_eInit()**.

Parent topic:ZTIMER_eOpen

Parameters

- *pu8TimerIndex* Pointer to location containing the index number of the timer in the list of timers initialized using **ZTIMER_eInit()**
- *pfCallback* Pointer to the user-defined callback function for the timer

- *pvParams* Pointer to a list of parameter values for the timer *u8Flags* Flag indicating whether the timer should allow or prevent sleep, one of:
  - ZTIMER_FLAG_ALLOW_SLEEP
  - ZTIMER_FLAG_PREVENT_SLEEP

Parent topic:ZTIMER_eOpen

Returns

- E_ZTIMER_OK (timer successfully opened)
- E_ZTIMER_FAIL (timer not opened)

Parent topic:ZTIMER_eOpen

Parent topic: ZigBee Timer functions

ZTIMER_eClose

```
ZTIMER_teStatus ZTIMER_eClose(uint8 u8TimerIndex);
```

Description This function is used to close the specified software timer when it is no longer needed. The timer must have been previously opened using **ZTIMER_eOpen()**.

Parent topic:ZTIMER_eClose

Parameters *u8TimerIndex:* Index number of the timer in the list of timers initialized using **ZTIMER_eInit()**.

Parent topic:ZTIMER_eClose

Returns

- E_ZTIMER_OK (timer successfully closed)
- E_ZTIMER_FAIL (timer not closed - may be running or already closed)

Parent topic:ZTIMER_eClose

Parent topic: ZigBee Timer functions

ZTIMER_eStart

```
ZTIMER_teStatus ZTIMER_eStart(uint8 u8TimerIndex,
                              uint32 u32Time);
```

Description This function is used to start the specified software timer. The length of time for which the timer will run must be specified in milliseconds.

Before a timer is started, it must have been opened using **ZTIMER_eOpen()**. Once started, the timer can be stopped (before it expires) using **ZTIMER_eStop()**.

Parent topic:ZTIMER_eStart

Parameters

- *pu8TimerIndex:* Index number of the timer in the list of timers initialized using **ZTIMER_eInit()**.
- *u32Time:* The time, in milliseconds, for which the timer should run.

Parent topic:ZTIMER_eStart

Returns

- E_ZTIMER_OK (timer successfully started)
- E_ZTIMER_FAIL (timer not started)

Parent topic:ZTIMER_eStart

Parent topic: ZigBee Timer functions

ZTIMER_eStop

```
ZTIMER_teStatus ZTIMER_eStop(uint8 u8TimerIndex);
```

Description This function is used to stop the specified software timer (before it expires). The timer must have been previously started using **ZTIMER_eStart()**.

Parent topic:ZTIMER_eStop

Parameters *pu8TimerIndex:* Index number of the timer in the list of timers initialized using **ZTIMER_eInit().**

Parent topic:ZTIMER_eStop

Returns

- E_ZTIMER_OK (timer successfully stopped)
- E_ZTIMER_FAIL (timer not stopped - may be already stopped or expired)

Parent topic:ZTIMER_eStop

Parent topic: ZigBee Timer functions

ZTIMER_eGetState

```
ZTIMER_teStatus ZTIMER_eGetState(uint8 u8TimerIndex);
```

Description This function is used to obtain the current state of the specified software timer. The possible reported states are:

- Running
- Stopped
- Expired
- Closed

Parent topic:ZTIMER_eGetState

Parameters *pu8TimerIndex* Index number of the timer in the list of timers initialized using **ZTIMER_eInit()**.

Parent topic:ZTIMER_eGetState

Parent topic: ZigBee Timer functions

Parent topic: ZigBee Timer resources

ZigBee timer structures

ZTIMER_tsTimer The ZigBee timer structure is shown below. It is used to represent a single software timer that may be used by the application.

```
typedef struct
{
    ZTIMER_teState      eState;
    uint32              u32Time;
    void               *pvParameters;
    ZTIMER_tpfCallback pfCallback;
} ZTIMER_tsTimer;
```

where:

- eStateerepresents the current state of the timer, as one of:
  - E_ZTIMER_STATE_CLOSED
  - E_ZTIMER_STATE_STOPPED
  - E_ZTIMER_STATE_RUNNING
  - E_ZTIMER_STATE_EXPIRED
- u32Timeis the remaining time, in milliseconds, that the timer still has to run
- pvParametersis a pointer to a set of parameters used by the timer
- pfCallbackis a pointer to the user-defined callback function that will be called when the timer expires - this function has the prototype:

  ```
  typedef void \(\*ZTIMER\_tpfCallback\)\(void \pvParam\*\);
  ```

  where pvParam is a pointer to the timer parameters.

Parent topic: ZigBee timer structures

Parent topic: ZigBee Timer resources

Parent topic:*General ZPS Resources*

Critical Section and Mutex Resources The Critical Section and Mutex functions are concerned with protecting sections of application code from preemption and re-entrancy. These resources are provided in the header file **portmacro.h**.

- The Critical Section and Mutex functions are described in Section 10.3.1.
- The Critical Section and Mutex structures are described in Section 10.3.2.

Critical Section and Mutex functions The functions are listed below.

Function page

1. ZPS_eEnterCriticalSection
2. ZPS_eExitCriticalSection
3. ZPS_u8GrabMutexLock
4. ZPS_u8ReleaseMutexLock

Parent topic: Critical Section and Mutex functions

ZPS_eEnterCriticalSection

```
uint8 ZPS_eEnterCriticalSection(
                void *hMutex,
                uint32* psIntStore);
```

Description This function can be used to mark the end of a critical section of application code. The function **ZPS_eEnterCriticalSection()** must have been called at the start of the critical section.

This function can be used to mark the start of a critical section of application code - this is a code section that cannot be preempted by an interrupt with priority level less than 12. The function **ZPS_eExitCriticalSection()** must be called at the end of the critical section.

A pointer to a 'priority level' value must be provided, which contains the current priority level of the main application thread (when critical sections are not being executed). When a critical section is entered, the priority level of the main thread is increased such that interrupts with a priority of 11 or less cannot preempt the main thread. At the end of the critical section, the priority level of the main thread is returned to its value from before the critical section was entered.

Optionally, a mutex can also be applied during the critical section to protect the section from re-entrancy. If a mutex is required, a pointer must be provided to a user- defined mutex function with the following prototype:

((bool_t*) (*) (void))

This function must define and maintain a Boolean flag that indicates whether the corresponding mutex is active (TRUE) or inactive (FALSE). This flag is used by **ZPS_eEnterCriticalSection()** to determine whether the mutex is available.

- If this flag reads as FALSE, the mutex is applied and the above mutex function must set the flag to TRUE.
- If the flag is already TRUE, then the mutex cannot be applied - in this case, **ZPS_eEnterCriticalSection()** returns a failure.

Critical sections and mutexes are further described in Section 6.9.3.

Parent topic:ZPS_eEnterCriticalSection

Parameters

- *hMutex* Pointer to user-defined mutex function (see above) - set to NULL if no mutex is required
- *psIntStore* Pointer to structure containing 'priority level' value (see Section 10.3.2.1)

Parent topic:ZPS_eEnterCriticalSection

Returns 0x00 for success, 0x01 for failure (all other values are reserved)

Parent topic:ZPS_eEnterCriticalSection

Parent topic: Critical Section and Mutex functions

ZPS_eExitCriticalSection

```
uint8 ZPS_eExitCriticalSection(
                void *hMutex,
                uint32* psIntStore);
```

Description This function can be used to mark the end of a critical section of application code. The function **ZPS_eEnterCriticalSection()** should be called at the start of the critical section.

A pointer to the 'priority level' value must be provided. If a mutex was used in the critical section, a pointer to the relevant mutex function must be provided in order to release the mutex.

Critical sections and mutexes are further described in Section 6.9.3.

Parent topic:ZPS_eExitCriticalSection

Parameters

- *hMutex* Pointer to user-defined mutex function (see above) - set to`NULL` if no mutex was used.
- *psIntStore* Pointer to structure containing 'priority level' value (see Section 10.3.2.1).

Parent topic:ZPS_eExitCriticalSection

Returns 0x00 for success, 0x01 for failure (all other values are reserved)

Parent topic:ZPS_eExitCriticalSection

Parent topic: Critical Section and Mutex functions

ZPS_u8GrabMutexLock

```
uint8 ZPS_u8GrabMutexLock(
                void *hMutex,
                uint32* psIntStore);
```

Description This function can be used to apply a mutex at the start of a section of application code that is to be protected from re-entrancy. The function **ZPS_u8ReleaseMutexLock()**must be called at the end of the mutex-protected section to release the mutex.

A pointer must be provided to a user-defined mutex function with the following prototype:

((bool_t*) (*) (void))

This function must define and maintain a Boolean flag which indicates whether the corresponding mutex is active (TRUE) or inactive (FALSE). This flag is used by **ZPS_u8GrabMutexLock()** to determine whether the mutex is available. If this flag reads as FALSE, the mutex is applied and the above mutex function must set the flag to TRUE, but if the flag is already TRUE then the mutex cannot be applied - in the latter case, **ZPS_u8GrabMutexLock()** returns a failure.

A pointer to a 'priority level' value must be provided, which contains the current priority level of the main application thread (when mutex protection is not being implemented). When a mutex is applied, the priority level of the main thread is increased such that interrupts with a priority

of 11 or less cannot preempt the main thread. When the mutex is released, the priority level of the main thread will be returned to its value from before the mutex was applied.

Mutexes are further described in Section 6.9.3.

Parent topic:ZPS_u8GrabMutexLock

Parameters

- *hMutex* Pointer to user-defined mutex function (see above)
- *psIntStore* Pointer to structure containing 'priority level' value (see Section 10.3.2.1).

Parent topic:ZPS_u8GrabMutexLock

Returns 0x00 for success, 0x01 for failure (all other values are reserved).

Parent topic:ZPS_u8GrabMutexLock

Parent topic: Critical Section and Mutex functions

ZPS_u8ReleaseMutexLock

```
uint8 ZPS_u8ReleaseMutexLock(
                void *hMutex,
                uint32* psIntStore);
```

Description This function can be used to release a mutex that has been applied to a section of application code. The function **ZPS_u8GrabMutexLock()** must have been called at the start of the mutex-protected section.

A pointer to the relevant mutex function must be provided in order to release the mutex. A pointer to the 'priority level' value must also be provided.

Mutexes are further described in Section 6.9.3.

Parent topic:ZPS_u8ReleaseMutexLock

Parameters

- *hMutex* Pointer to user-defined mutex function (see above).
- *psIntStore* Pointer to structure containing 'priority level' value (see Section 10.3.2.1).

Parent topic:ZPS_u8ReleaseMutexLock

Returns 0x00 for success, 0x01 for failure (all other values are reserved)

Parent topic:ZPS_u8ReleaseMutexLock

Parent topic: Critical Section and Mutex functions

Parent topic: Critical Section and Mutex Resources

Critical Section and Mutex Structures

u32MicroIntStorage u32MicroIntStorageis a 32-bit mask of the interrupts currently enabled. The u32MicroIntStoragestructure is used in critical sections and mutex-protected sections where u32MicroIntStoragetakes the interrupts that have been enabled.

Parent topic: Critical Section and Mutex Structures

Parent topic: Critical Section and Mutex Resources

Parent topic:*General ZPS Resources*

Event and Status Codes This chapter summarizes the event and return/status codes of the ZigBee PRO stack.

Events The events that can be generated by the ZigBee PRO stack are enumerated in the structure ZPS_teAfEventType(from the AF API), shown below.

```
typedef enum {
ZPS_EVENT_NONE,                                    /* 0, 0x00 */
ZPS_EVENT_APS_DATA_INDICATION,                     /* 1, 0x01 */
ZPS_EVENT_APS_DATA_CONFIRM,                        /* 2, 0x02 */
ZPS_EVENT_APS_DATA_ACK,                            /* 3, 0x03 */
ZPS_EVENT_NWK_STARTED,                             /* 4, 0x04 */
ZPS_EVENT_NWK_JOINED_AS_ROUTER,                    /* 5, 0x05 */
ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE,                 /* 6, 0x06 */
ZPS_EVENT_NWK_FAILED_TO_START,                     /* 7, 0x07 */
ZPS_EVENT_NWK_FAILED_TO_JOIN,                      /* 8, 0x08 */
ZPS_EVENT_NWK_NEW_NODE_HAS_JOINED,                 /* 9, 0x09 */
ZPS_EVENT_NWK_DISCOVERY_COMPLETE,                  /* 10, 0x0a */
ZPS_EVENT_NWK_LEAVE_INDICATION,                    /* 11, 0x0b */
ZPS_EVENT_NWK_LEAVE_CONFIRM,                       /* 12, 0x0c */
ZPS_EVENT_NWK_STATUS_INDICATION,                   /* 13, 0x0d */
ZPS_EVENT_NWK_ROUTE_DISCOVERY_CONFIRM,             /* 14, 0x0e */
ZPS_EVENT_NWK_POLL_CONFIRM,                        /* 15, 0x0f */
ZPS_EVENT_NWK_ED_SCAN,                             /* 16, 0x10 */
ZPS_EVENT_ZDO_BIND,                                /* 17, 0x11 */
ZPS_EVENT_ZDO_UNBIND,                              /* 18, 0x12 */
ZPS_EVENT_ZDO_LINK_KEY,                            /* 19, 0x13 */
ZPS_EVENT_BIND_REQUEST_SERVER,                     /* 20, 0x14.*/
ZPS_EVENT_ERROR,                                   /* 21, 0x15 */
ZPS_EVENT_APS_INTERPAN_DATA_INDICATION,            /* 22, 0x16 */
ZPS_EVENT_APS_INTERPAN_DATA_CONFIRM,               /* 23, 0x17 */
ZPS_EVENT_APS_ZGP_DATA_INDICATION,                 /* 24, 0x18 */
ZPS_EVENT_APS_ZGP_DATA_CONFIRM,                    /* 25, 0x19 */
ZPS_EVENT_TC_STATUS,                               /* 26, 0X1A */
ZPS_EVENT_NWK_DUTYCYCLE_INDICATION,                /* 27, 0x1B */
ZPS_EVENT_NWK_FAILED_TO_SELECT_AUX_CHANNEL,        /* 28, 0x1C */
ZPS_EVENT_NWK_ROUTE_RECORD_INDICATION,             /* 29, 0x1D */
ZPS_EVENT_NWK_FC_OVERFLOW_INDICATION,              /* 30, 0x1E */
ZPS_ZCP_EVENT_FAILURE
} ZPS_teAfEventType;
```

The events in the above structure are outlined in the table below.

Note: The AF structures which contain the data for the above events are detailed in Section 8.2.2, Event Structures .

ZigBee PRO stack Events

Stack Event	Description
ZPS_EVENT_N	Used as initial value in structure which receives a message collected from a message queue.
ZPS_EVENT_A	Indicates that data has arrived on the local node. The event provides information about the data packet through the structure ZPS_tsAfDataIndEvent- see Section 8.2.2.3.
ZPS_EVENT_A	Indicates whether a sent data packet has been successfully passed down the stack and has reached the next hop node toward its destination. The results are reported through the structure ZPS_tsAfDataConfEvent- see Section 8.2.2.4.
ZPS_EVENT_A	Indicates that a sent message has reached its destination node. Details of the received acknowledgment are reported through the structure ZPS_tsAfDataAckEvent- see Section 8.2.2.5.
ZPS_EVENT_N	Indicates that network has started on Coordinator. This is reported through the structure ZPS_tsAfNwkFormationEvent- see Section 8.2.2.6. 'Permit joining' state is set as specified in APL data structure.
ZPS_EVENT_N	Indicates that device has successfully joined network - as Router and reports allocated network address through the structure ZPS_tsAfNwkJoinedEvent- see Section 8.2.2.7.
ZPS_EVENT_N	Indicates that device has successfully joined network as End Device and reports allocated network address through the structure ZPS_tsAfNwkJoinedEvent- see Section 8.2.2.7.
ZPS_EVENT_N	Indicates that network has failed to start on Coordinator.
ZPS_EVENT_N	Indicates that device failed to join network. This is reported through the structure ZPS_tsAfNwkJoinFailedEvent- see Section 8.2.2.8
ZPS_EVENT_N	Indicates to Coordinator or Router that new node has joined as child and reports details of new child through the structure ZPS_tsAfNwkJoinIndEvent- see Section 8.2.2.10.
ZPS_EVENT_N	Indicates that network discovery on Router or End Device has finished and reports details of detected net- work through the structure ZPS_tsAfNwkDiscoveryEvent- see Section 8.2.2.9. This event (and associated structure) is generated for each network detected.
ZPS_EVENT_N	Indicates that a neighboring node has left the network or a remote node has requested the local node to leave. Details are provided through the structure ZPS_tsAfNwkLeaveIndEvent- see Section 8.2.2.11.
ZPS_EVENT_N	Reports the results of a node leave request issued by the local node. The results are reported through the structure ZPS_tsAfNwkLeaveConfEvent- see Section 8.2.2.12.
ZPS_EVENT_N	Reports network status event from a remote or local node through the structure ZPS_tsAfNwkStatusIndEvent- see Section 8.2.2.13.
ZPS_EVENT_N	Indicates that a route discovery has been performed. The results are reported in the structure ZPS_tsAfNwkRouteDiscoveryConfEvent- see Section8.2.2.14.
ZPS_EVENT_N	Generated on an End Device to indicate that a poll request submitted to its parent has completed. The outcome of the poll request is indicated through the structure ZPS_tsAfPollConfEvent- see Section8.2.2.15.
ZPS_EVENT_N	Indicates that an 'energy detect' scan in the 2.4-GHz radio band has completed. The results of the scan are reported through the structure ZPS_tsAfNwkEdScanConfEvent- see Section 8.2.2.16.
ZPS_EVENT_Z	Indicates that the local node has been successfully bound to one or more remote nodes. The details of the binding are reported through the structure ZPS_tsAfZdoBindEvent- see Section 8.2.2.18.
ZPS_EVENT_Z	Indicates that the local node has been successfully unbound from one or more remote nodes. The details of the unbinding are reported through the structure ZPS_tsAfZdoUnbindEvent- see Section 8.2.2.19.
ZPS_EVENT_Z	Indicates that a new application link key has been received and installed, and is ready for use. The details of the link key are reported through the structure ZPS_tsAfZdoLinkKeyEvent- see Section 8.2.2.20.
ZPS_EVENT_I	Indicates the results of a bound data transmission. The results are reported through the structure ZPS_tsAfBindRequestServerEvent- see Section 8.2.2.21.
ZPS_EVENT_I	Indicates that an error has occurred on the local node. The nature of the error is reported through the structure ZPS_tsAfErrorEvent- see Section 8.2.2.17.
ZPS_EVENT_A	Indicates that an inter-PAN communication has arrived (from a node in another network). Details of the inter-PAN communication are reported through the

Note: Event handling is outlined in "*Appendix A*".

Parent topic:*Event and Status Codes*

Return/Status Codes The return/status codes that can result from ZigBee PRO API function calls are divided into the following groups:

- ZDP codes - see Section 11.2.1
- APS codes - see Section 11.2.2
- NWK codes - see Section 11.2.3
- MAC codes - see Section 11.2.4
- Extended error codes - see Section 11.2.5

ZDP codes The ZDP codes are carried in request and response messages.

ZDP codes

Name	Valu	Description
ZPS_APL_ZDP_E_INV	0x8(	The supplied request type was invalid.
ZPS_APL_ZDP_E_DEV	0x81	The requested device did not exist on a device following a child descriptor request to a parent.
ZPS_APL_ZDP_E_INV.	0x82	The supplied endpoint was equal to 0x00 or between 0xF1 and 0xFF.
ZPS_APL_ZDP_E_NOT	0x83	The requested endpoint is not described by a Simple descriptor.
ZPS_APL_ZDP_E_NOT	0x84	The requested optional feature is not supported on the target device.
ZPS_APL_ZDP_E_TIM	0x85	A timeout has occurred with the requested operation.
ZPS_APL_ZDP_E_NO_	0x86	The End Device bind request was unsuccessful due to a failure to match any suitable clusters.
ZPS_APL_ZDP_E_NO_	0x88	The unbind request was unsuccessful due to the Co- ordinator or source device not having an entry in its binding table to unbind.
ZPS_APL_ZDP_E_NO_	0x89	A child descriptor was not available following a discovery request to a parent.
ZPS_APL_ZDP_E_INS	0x8A	The device does not have storage space to support the requested operation.
ZPS_APL_ZDP_E_NOT	0x8E	The device is not in the proper state to support the requested operation.
ZPS_APL_ZDP_E_TAB	0x8(	The device does not have table space to support the operation.
ZPS_APL_ZDP_E_NOT	0x8I	The permissions configuration table on the target indicates that the request is not authorized from this device.

Parent topic: Return/Status Codes

APS codes The APS codes relate to sending/receiving messages.

APS codes

Name	Valu	Description
ZPS_APL_APS_E_ASD	0xA0	A transmit request failed since the ASDU is too large and fragmentation is not supported.
ZPS_APL_APS_E_DEF	0xA1	A received fragmented frame could not be defragmented at the current time.
ZPS_APL_APS_E_DEF	0xA2	A received fragmented frame could not be defragmented since the device does not support fragmentation.
ZPS_APL_APS_E_ILLI	0xA3	A parameter value was out of range.
ZPS_APL_APS_E_INV.	0xA4	An APSME-UNBIND.request failed due to the requested binding link not existing in the binding table.
ZPS_APL_APS_E_INV.	0xA5	An APSME-REMOVE-GROUP.request has been issued with a group identifier that does not appear in the group table.
ZPS_APL_APS_E_INV.	0xA6	A parameter value was invalid or out of range.
ZPS_APL_APS_E_NO_	0xA7	An APSDE-DATA.request requesting acknowledged transmission failed due to no acknowledgment being received.
ZPS_APL_APS_E_NO_	0xA8	An APSDE-DATA.request with a destination addressing mode set to 0x00 failed due to there being no devices bound to this device.
ZPS_APL_APS_E_NO_	0xA9	An APSDE-DATA.request with a destination addressing mode set to 0x03 failed due to no corresponding short address found in the address map table.
ZPS_APL_APS_E_NOT	0xAA	An APSDE-DATA.request with a destination addressing mode set to 0x00 failed due to a binding table not being supported on the device.
ZPS_APL_APS_E_SEC	0xAB	An ASDU was received that was secured using a link key.
ZPS_APL_APS_E_SEC	0xAC	An ASDU was received that was secured using a network key.
ZPS_APL_APS_E_SEC	0xAD	An APSDE-DATA.request requesting security has resulted in an error during the corresponding security processing.
ZPS_APL_APS_E_TAB	0xAE	An APSME-BIND.request or APSME.ADDGROUP.request issued when the binding or group tables, respectively, were full.
ZPS_APL_APS_E_UNS	0xAF	An ASDU was received without any security.
ZPS_APL_APS_E_UNS	0xB0	An APSME-GET.request or APSMESET. request has been issued with an unknown attribute identifier.

Parent topic: Return/Status Codes

NWK codes The NWK codes come from the NWK layer of the stack and may be returned by any ZigBee PRO API function with a non-void return.

NWK codes

Name	Valu	Description
ZPS_NWK_ENUM_SUCCESS	0x00	Success
ZPS_NWK_ENUM_INVALID_PA	0xC1	An invalid or out-of-range parameter has been passed
ZPS_NWK_ENUM_INVALID_RE	0xC2	Request cannot be processed
ZPS_NWK_ENUM_NOT_PERMI	0xC3	NLME-JOIN.request not permitted
ZPS_NWK_ENUM_STARTUP_FA	0xC4	NLME-NETWORK-FORMATION.request failed
ZPS_NWK_ENUM_ALREADY_PI	0xC5	NLME-DIRECT-JOIN.request failure - device already present
ZPS_NWK_ENUM_SYNC_FAILU	0xC6	NLME-SYNC.request has failed
ZPS_NWK_ENUM_NEIGHBOR_	0xC7	NLME-DIRECT-JOIN.request failure - no space in Router table
ZPS_NWK_ENUM_UNKNOWN_	0xC8	NLME-LEAVE.request failure - device not in Neighbor table
ZPS_NWK_ENUM_UNSUPPORT	0xC9	NLME-GET/SET.request unknown attribute identifier
ZPS_NWK_ENUM_NO_NETWO	0xCA	NLME-JOIN.request detected no networks
ZPS_NWK_ENUM_RESERVED_1	0xCB	Reserved
ZPS_NWK_ENUM_MAX_FRM_C	0xCC	Security processing has failed on outgoing frame due to maximum frame counter
ZPS_NWK_ENUM_NO_KEY	0xCD	Security processing has failed on outgoing frame due to no key
ZPS_NWK_ENUM_BAD_CCM_O	0xCE	Security processing has failed on outgoing frame due CCM
ZPS_NWK_ENUM_NO_ROUTIN	0xCF	Attempt at route discovery has failed due to lack of table space
ZPS_NWK_ENUM_ROUTE_DISC	0xD0	Attempt at route discovery has failed due to any reason except lack of table space
ZPS_NWK_ENUM_ROUTE_ERR	0xD1	NLDE-DATA.request has failed due to routing failure on sending device
ZPS_NWK_ENUM_BT_TABLE_F	0xD2	Broadcast or broadcast-mode multicast has failed as there is no room in BTT
ZPS_NWK_ENUM_FRAME_NOT	0xD3	Unicast mode multi-cast frame was discarded pending route discovery
ZPS_NWK_ENUM_FRAME_IS_B	0xD4	Unicast frame does not have a route available but it is buffered for automatic resend.

Parent topic: Return/Status Codes

MAC codes The MAC codes come from the IEEE 802.15.4 MAC layer of the stack. The MAC codes may be returned by any ZigBee PRO API function with a non-void return. The codes are also described in the *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*.

MAC codes

Name	Value	Description
MAC_ENUM_SUCCESS	0x00	Success
MAC_ENUM_BEACON_LOSS	0xE0	Beacon loss after synchronization request
MAC_ENUM_CHANNEL_ACCESS_I	0xE1	CSMA/CA channel access failure
MAC_ENUM_DENIED	0xE2	GTS request denied
MAC_ENUM_DISABLE_TRX_FAILU	0xE3	Could not disable transmit or receive
MAC_ENUM_FAILED_SECURITY_C	0xE4	Incoming frame failed security check
MAC_ENUM_FRAME_TOO_LONG	0xE5	Frame too long, after security processing, to be sent
MAC_ENUM_INVALID_GTS	0xE6	GTS transmission failed
MAC_ENUM_INVALID_HANDLE	0xE7	Purge request failed to find entry in queue
MAC_ENUM_INVALID_PARAMETI	0xE8	Out-of-range parameter in function
MAC_ENUM_NO_ACK	0xE9	No acknowledgment received when expected
MAC_ENUM_NO_BEACON	0xEA	Scan failed to find any beacons
MAC_ENUM_NO_DATA	0xEB	No response data after a data request
MAC_ENUM_NO_SHORT_ADDRES	0xEC	No allocated network (short) address for operation
MAC_ENUM_OUT_OF_CAP	0xED	Receiver-enable request could not be executed, as CAP finished
MAC_ENUM_PAN_ID_CONFLICT	0xEE	PAN ID conflict has been detected
MAC_ENUM_REALIGNMENT	0xEF	Coordinator realignment has been received
MAC_ENUM_TRANSACTION_EXPI	0xF0	Pending transaction has expired and data discarded
MAC_ENUM_TRANSACTION_OVE	0xF1	No capacity to store transaction
MAC_ENUM_TX_ACTIVE	0xF2	Receiver-enable request could not be executed, as in transmit state
MAC_ENUM_UNAVAILABLE_KEY	0xF3	Appropriate key is not available in ACL
MAC_ENUM_UNSUPPORTED_ATT	0xF4	PIB Set/Get on unsupported attribute

Parent topic: Return/Status Codes

Extended error codes If extended error handling is implemented (see Section 6.7), it provides more detail about the error that led to any one of the following function return codes:

- APS codes 0xA3, 0xA6, and 0xAD (see Section 11.2.2).
- NWK code 0xC2 (see Section 11.2.3).

The extended error codes, which elaborate on the above codes are provided in the ZPS_teExtendedStatusenumerations.

Extended error codes

Name	Value	Description
ZPS_XS_OK	0x00	Success
ZPS_XS_E_FATAL	0x01	Fatal error - retrying will cause the error a
ZPS_XS_E_LOOPBACK_BAD_ENDPOINT	0x02	Endpoint is not valid for loopback (fatal er
ZPS_XS_E_SIMPLE_DESCRIPTOR_NO_ OUTPUT_CLUSTER	0x03	No output cluster in the Simple descriptor
ZPS_XS_E_FRAG_NEEDS_ACK	0x04	Fragmented data requests must be sent wi
ZPS_XS_E_COMMAND_MANAGER_BAD_ PARAMETER	0x05	Bad parameter has been passed to the com
ZPS_XS_E_INVALID_ADDRESS	0x06	Address parameter is out-of-range (fatal er
ZPS_XS_E_INVALID_TX_ACK_FOR_LOCAL_EP	0x07	TX ACK bit has been set when attempting t
ZPS_XS_E_RESOURCE	0x08	Resource error/shortage - retrying may su
ZPS_XS_E_NO_FREE_NPDU	0x80	No free NPDUs (resource error) - the numb
ZPS_XS_E_NO_FREE_APDU	0x81	No free APDUs (resource error) - the numb
ZPS_XS_E_NO_FREE_SIM_DATA_REQ	0x82	No free simultaneous data request handle
ZPS_XS_E_NO_FREE_APS_ACK	0x83	No free APS acknowledgment handles (res
ZPS_XS_E_NO_FREE_FRAG_RECORD	0x84	No free fragment record handles (resource

Name	Value	Description
ZPS_XS_E_NO_FREE_MCPS_REQ	0x85	No free MCPS request descriptors (resourc
ZPS_XS_E_NO_FREE_LOOPBACK	0x86	Loopback send is currently busy (resource
ZPS_XS_E_NO_FREE_EXTENDED_ADDR	0x87	No free entries in the extended address ta
ZPS_XS_E_SIMPLE_DESCRIPTOR_NOT_ FOUND	0x88	Simple descriptor does not exist for this en
ZPS_XS_E_BAD_PARAM_APSDE_REQ_RSP	0x89	Bad parameter has been found while proc
ZPS_XS_E_NO_RT_ENTRY	0x8a	No routing table entries free
ZPS_XS_E_NO_BTR	0x8b	No Broadcast transaction table entries free
ZPS_XS_E_FRAME_COUNTER_ERROR	0xC0	Decryption failed due to frame counter of
ZPS_XS_E_CCM_INVALID_ERROR	0xC1	Decryption failed due to invalid CCM data.
ZPS_XS_E_UNKNOWN_SRC_ADDR	0xC2	Decryption failed due to unknown source
ZPS_XS_E_NO_KEY_DESCRIPTOR	0xC3	Decryption failed due to missing the match
ZPS_XS_E_NULL_KEYDESCR	0xC4	Decryption failed due to NULL key descrip
ZPS_XS_E_PDUM_ERROR	0xC5	Decryption failed due to PDUM packet clon
ZPS_XS_E_NULL_EXT_ADDR	0xC6	Encryption failed due to missing the Exter
ZPS_XS_E_ENCRYPT_NULL_DESCR	0xC7	Encryption failed due to NULL key descrip
ZPS_XS_E_ENCRYPT_FRAME_COUNTER_FAIL	0xC8	Encryption failed due to frame counter of
ZPS_XS_E_ENCRYPT_DEFAULT	0xC9	Encryption failed due to internal error.
ZPS_XS_E_FRAME_COUNTER_EXPIRED	0xCA	Decryption failed due to frame counter ex

Parent topic: Return/Status Codes

Parent topic:*Event and Status Codes*

ZigBee network parameters This chapter lists and describes the ZigBee network parameters that can be set using the ZPS Configuration Editor described in *Chapter 13*.

Basic parameters The basic parameters are listed and described in the table below.

ZigBee Wireless Network parameters

Parameter Name	Description	Default Value	Range
Default Extend Pan ID	The default Extended PAN ID (EPID) when add- ing new devices to the wireless network. The extended PAN ID is the globally unique 64-bit identifier for the network. This identifier is used to avoid PAN ID conflicts between distinct networks and must be unique among the networks overlap- ping in a given area. If the value is zero on the Coordinator, the device will use its own IEEE/ MAC address as the EPID. A zero value on a Router/End Device means that the device will not look for a particular EPID when joining a network. Note that this value is the default EPID used when adding devices in the ZPS Configuration Editor. The actual EPID used for an individual device can be set via the parameter *APS Use Extended PAN ID*– see Section 12.7.	0	64 bits
Default Security Enabled	The default setting for Security Enabled when adding new devices to the wireless network.	true	true / false
Maximum Number of Nodes	The maximum number of nodes for the wireless network. This setting controls the size of tables when adding new devices to the network to ensure adequate resources are available for correct operation a network of the specified size.	20	

The rest of the network parameters are detailed in the sections that follow, according to their area of application.

Parent topic:*ZigBee network parameters*

Profile definition parameters

Profile definition parameters

Parameter Name	Description	Default Value	Range
Profile Id	The application profile identifier. This is assigned by the ZigBee Alliance for a public profile.		16 bits(Value 0 is reserved for the ZDP, 0xFFFF is wildcard profile)
Name	Textual name for the profile. This is used as a prefix for generated macro definitions in **zps_gen.h**.		Valid C identifier. ("ZDP" is reserved for the ZigBee Device Profile)

Parent topic:*ZigBee network parameters*

Cluster definition parameters

Parameter Name	Description	Default Value	Range
Cluster Id	The cluster identifier. This may be defined by the ZigBee Alliance for a public cluster or may be manufacturer-specific.Clusters are designated as inputs or outputs in the simple descriptor for use in creating a binding table.	-	16 bits
Name	Textual name for the cluster. This is used as a prefix for generated macro definitions in **zps_gen.h**.	-	Valid C identifier

Parent topic:*ZigBee network parameters*

Coordinator parameters

Parameter Name	Description	Default Value	Range
Miscellaneous Coordinator Parameters			
Name	Textual name for the node. Used as a prefix when generating macro definitions in **zps_gen.h**.		Valid C identifier
Permit Joining Time	Default number of seconds for which permit joining is enabled.- '255' means permanently on- '0' means permanently off	255	0-255
Security Enabled	Specifies whether the Coordinator will secure communication with other devices in the network.	true	true / false
Initial Security Key	The initial key that will be used when security is enabled. These are selected from the keys available on the Trust Centre.	None	

Parent topic:*ZigBee network parameters*

Router parameters

Router Node Type Parameters

Parameter Name	Description	Default Value	Range
Miscellaneous Router Parameters			
Name	Textual name for the node. Used as a pre- fix when generating macro definitions in **zps_gen.h**.		Valid C identifier
Permit Joining Time	Default number of seconds for which permit joining is enabled.- 255 means permanently on- 0 means permanently off	255	0-255
Scan Duration Time	The length of time to scan the selected RF channels when searching for a network to join.The time spent scanning each channel is:[aBaseSuperframeDuration x (2n + 1)] symbolswhere n is the value of the Scan Duration Time parameter.	3	0 – 14
Security Enabled	Specifies whether the Router will secure communication with other devices in the network.	true	true / false
Initial Security Key	The initial key that will be used when security is enabled. These are selected from the keys available on the Trust Centre.	None	

Parent topic:*ZigBee network parameters*

End Device parameters **End Device Node Type Parameters**

Parameter Name	Description	Default Value	Range
Miscellaneous End Device Parameters			
Name	Textual name for the node. Used as a prefix when generating macro definitions in **zps_- gen.h**.		Valid C identifier
Scan Duration Time	The length of time to scan the selected RF channels when searching for a network to join. The time spent scanning each channel is: $(\text{aBaseSuperframeDuration} * (2n + 1))$ symbols where n is the value of the Scan Duration Time parameter.	3	0 – 14
Security Enabled	Specifies whether the End Device will secure communication with other devices in the network.	true	true / false
Initial Security Key	The initial key that will be used when security is enabled. These are selected from the keys available on the Trust Centre.	None	
Sleeping	Indicates whether the device will turn its receiver off and enter a low-power mode. The End Device's parent will buffer any incoming data until the device returns to its normal operating state and issues a poll request.	false	true / false
Number of Poll Failures Before Rejoin	This parameter controls the number of consecutive poll failures from when the device returns to its normal operating state before attempting to find a new parent by initiating a network rejoin.	5	0 will disable this behavior

Parent topic:*ZigBee network parameters*

Advanced device parameters These are advanced parameters for Coordinator, Router, and End Device.

Advanced device parameters

Parameter Name	Description
AF Parameters	
-	
AIB Parameters	
APS Designated Coordinator(read only)	Indicates that on start-up the no
APS Use Extended PAN ID	Indicates the Extended PAN ID (
APS Inter-frame Delay	Number of milliseconds betwee
APS Max Window Size	APS fragmented data window si
APS Non-member Radius	Multicast non-member radius si
APS Security Timeout Period	Authentication timeout period i
APS Use Insecure Join	Controls action when a secured
APS Layer Configuration Parameters	
APS Duplicate Table Size	The size of the APS layer duplica
APS Persistence Time	Time, in milliseconds, for which
Maximum Number of Simultaneous Data Requests	The maximum number of simul
Maximum Number of Simultaneous Data Requests with Acks	The maximum number of simul
Inter PAN	True if inter PAN functionality i
APS Poll Period	The polling period, in millisecon
Maximum Number of Received Simultaneous Fragmented Messages	Maximum number of simultane

Parameter Name	Description
Maximum Number of Transmitted Simultaneous Fragmented Messages	Maximum number of simultane
Network Layer Configuration Parameters for Coordinator and Routers	
Active Neighbor Table Size	Size of the active Neighbor table
Child Table Size	Size of the persisted sub-table o
Address Map Table Size	Size of the address map, which
Broadcast Transaction Table Size	Size of broadcast transaction tal
Discovery Neighbor Table Size	Size of the Discovery Neighbor t
Route Discovery Table Size	Size of the Route Discovery tabl
Route Record Table Size	Size of the Route Record table.
Routing Table Size	Size of the Routing table. This ta
Security Material Sets	Number of supported network
Network Layer Configuration Parameters for End Devices	
Active Neighbor Table Size	Size of the active Neighbor table
Address Map Table Size	Size of the address map, which
Broadcast Transaction Table Size	Size of broadcast transaction tal
Discovery Neighbor Table Size	Size of the Discovery Neighbor t
Route Discovery Table Size	Not applicable - set to one.
Route Record Table Size	Not applicable - set to one.
Routing Table Size	Not applicable - set to one.
Security Material Sets	Number of supported network
Stack Profile	The ZigBee Stack Profile which

Endpoint parameters **Endpoint parameters**

Parameter Name	Description	Default Value	Range
Application Device Id	Device ID for the endpoint.		
Application Device Version	Version number for the device.		
Enabled	Whether the endpoint is enabled or disabled.	true	true / false
End Point Id	The endpoint number (must be unique within the network).		1-240
Name	Textual name for the endpoint. Used as a prefix when generating macro definitions.		Valid C identifier
Profile	The application profile for the endpoint. This as a link to a profile definition.		

Input Cluster

Specifies that the endpoint will receive for the specified cluster.

Input cluster parameters

Parameter Name	Description	Default Value	Range
Cluster	A link to an input cluster that will receive on the endpoint.		
Receive APDU	A link to an APDU that will buffer any incoming messages.		
Discoverable	Defines whether the input cluster will be present in the endpoints simple descriptor which is used for service discovery.	true	true / false

Output cluster

Specifies that the endpoint will transmit for the specified cluster.

Output cluster parameters

Parameter Name	Description	Default Value	Range
Cluster	A link to an output cluster that will transmit on the endpoint.	-	-
Transmit APDUs	List of APDUs that will be used to transmit the cluster.	-	-
Discoverable	Defines whether the input cluster will be present in the endpoints Simple descriptor which is used for service dis- covery.	true	true / false

Parent topic: Advanced device parameters

Bound addressing table Specifies that the device should include a Binding table. Binding is optional. If Binding tables are used, they are located on any node which is a source for a binding, but the ZigBee Coordinator handles end device bind requests on behalf of all devices in the network. Nodes that use Binding tables should be allocated enough Binding table entries to handle their own communication needs.

Parameter Name	Description	Default Value	Range
Size	The size of the Binding table. Each binding table entry contains:- The node address and endpoint number of the source of the binding- The node address and endpoint number of the destination of the binding- The cluster ID for the bindingIf a binding is one-to-many, then a table entry is required for each destination.		

Parent topic: Advanced device parameters

PDU Manager The Protocol Data Unit Manager (PDUM) configuration is mandatory and must always be present.

PDU Manager parameters

Parameter Name	Description	Default Value	Range
Number of NPDUs	The number of NPDUs available to the ZigBee stack. These are internal to the stack.	16	8 or higher

APDU

Specifies a buffer to contain instances of a cluster.

APDU parameters

Parameter Name	Description	Default Value	Range
Instances	The maximum number of instances of this APDU. Note that this value must be greater than the value of the parameter *Maximum Number of Simultaneous Data Requests with Acks* - see Advanced device parameters.		
Name	The name of the APDU. This is the identifier that should be used in the application C code to refer to the APDU.		Valid C identifier
Size	The maximum size of the APDU.		

Parent topic: Advanced device parameters

Group Addressing table Specifies that the device contains a Group table.

Group Addressing table parameters

Parameter Name	Description	Default Value	Range
Size	The size of the Group table. Group membership for endpoints on the current device is controlled by adding and removing entries in the Group table.	-	-

Parent topic: Advanced device parameters

RF channels Specifies the default RF channels that the device will operate on. If not present, the default will be all channels.

RF channel parameters

Parameter Name	Description	Default Value	Range
Channel x(x=11-26)	Control for channel x – setting to true includes the channel in energy scan. By default, only channel 15 is included.	true for x=15, false for all other values	true / false

Parent topic: Advanced device parameters

MAC interface table Specifies the number of interfaces which are present on this device. One interface must be setup.

MAC interface table parameters

Parameter Name	Description	Default Value	Range
Channel List size (Needs to be set to 1 if only 2.4g)	Number of channel masks	1	No value ranges
Enabled	Is interface enabled/disabled	Enabled	TRUE/FALSE
Index	Index of the entry	0	0 to channel list size
Radio Type	2.4G or SubGig	RT2400MH	RT2400MHz/ RT868MHz/ RT915MHz
Routers Allowed	Are routers allowed on this interface	FALSE	TRUE/FALSE

Parent topic: Advanced device parameters

Node descriptor This is mandatory and defines the type and capabilities of the node.

Node descriptor parameters

Parameter Name	Description	Default Value	Range
Descriptor Availability Parameters			
Complex Descriptor Available	Complex descriptors are not supported. Not editable.	false	false
User Descriptor Available	Indicates whether a user descriptor is present. Not editable.	false	true / false
Descriptor Capabilities Parameters			
Extended Active Endpoint List Available	Indicates whether an extended active endpoint list is available. Not editable.	false	false
Extended Simple Descriptor List Available	Indicates whether an extended simple descriptor list is available. Not editable.	false	false
MAC Capability Flags			
Allocate Address	Indicates whether the device allocates short (network) addresses or not. Not editable.		true / false
Alternate PAN Coordinator	Indicates whether the device acts as an alternative PAN Coordinator. Not editable.		true / false
Device type	Indicates whether the device is a Full Functionality Device (FFD) or Reduced Functionality Device (RFD). Not editable.		true / false
Power source	Indicates whether the device is mains powered or not. Not editable.		true / false
Rx On When Idle	Indicates whether the device has its receiver enabled while the device is idle. Not editable.		true / false
Security	Indicates whether the device uses high or standard security. Only standard security is supported. Not editable.	false	true / false
Miscellaneous parameters			
APS flags	Not editable.	0	0
Frequency Band	Frequency band of radio. The the JN518x and K32W041/K32W061/K32W1/MCXW71/ MCXW72/RW612 devices only support the 2.4 GHz band. Not editable.	2.4 GHz	2.4 GHz
Logical Type	The device type: Coordinator, Router, or End Device. Not editable.		ZC/ZR/ZED
Manufacturer Code	The manufacturer ID code. The ZigBee Alliance allocates this code.		0 - 65535
Maximum buffer size	The maximum buffer size. Not editable.	127	
Maximum incoming transfer size	The maximum incoming transfer size supported by the device. This is calculated from the APDU sizes for input clusters. Not editable.		
Maximum outgoing transfer size	The maximum incoming transfer size supported by the device. This is calculated from the APDU sizes for output clusters. Not editable		
System Server Capabilities parameters			
Backup binding	Indicates if the node can act as a back-up binding table	false	true

Parent topic: Advanced device parameters

Node Power Descriptor The Node Power descriptor for the device is mandatory.

Node Power Descriptor Parameters

Parameter Name	Description	Default Value	Range
Available Power Sources parameters			
Constant power	Indicates whether a constant power source is available.	false	true / false
Disposable Battery	Indicates whether a disposable battery power source is available.	false	true / false
Rechargeable Battery	Indicates whether a rechargeable battery power source is available.	false	true / false
Miscellaneous parameters			
Default power mode	The default power mode of the device.	Synchronized with RxOnWhenIdle	Synchronized with RxOnWhenIdle / Periodic / Constant Power
Default power source	The default power source of the device.	Constant / rechargeable / disposable	Constant

Parent topic: Advanced device parameters

Key Descriptor table Specifies that the device should contain a Key Descriptor Table (for APS security).

Key Descriptor table parameters

Parameter Name	Description	Default Value	Range
Size	The size of the key descriptor table.		1 or higher

Preconfigured Key

Specifies a pre-configured link key for the Key Descriptor Table.

Preconfigured Key parameters

Parameter Name	Description	Default Value	Range
IEEE address	The IEEE address to use with the key.	-	64 bit
Key	The pre-configured key value.	-	128 bit

Parent topic: Advanced device parameters

Trust Centre Specifies that the device will have the capability to act as a Trust Centre.

Parameter Name	Description	Default Value	Range
Device Table Size	The size of the Trust Centre's device table.	Maximum Number of Nodes setting from the ZigBee PRO Wireless Network	1 or higher

A Trust Centre link key must be pre-set in the Key Descriptor Table (see Section12.7.9) on each node.

Parent topic: Advanced device parameters

Parent topic:*ZigBee network parameters*

ZDO configuration Specifies the ZigBee Device Object (ZDO) servers that are present on the device. Most of these are mandatory for a ZCP.

The ZDO configuration parameters are detailed in the following categories:

- **Category Page**
- *Default Server*
- *ZDO Client*
- *Device Annce Server*
- *Active Ep Server*
- *Nwk Addr Server*
- *IEEE Address Server*
- *System Server Discovery Server*
- *Permit Joining Server*
- *Node Descriptor Server*
- *Power Descriptor Server*
- *Match Descriptor Server*
- *Simple Descriptor Server*
- *Mgmt Lqi Server*
- *Mgmt Rtg Server*
- *Mgmt Leave Server*
- *Mgmt NWK Update Server*
- *Bind Unbind Server*
- *Extended Active Ep Server*
- *Extended Simple Descriptor Server*
- *End Device Bind Server*
- *Parent Announcement Server*
- *Management Enhanced Network Update Server*
- *MIB IEEE List Server*

Default Server

Mandatory. Replies to any unimplemented server requests.

Default Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to unimplemented server request messages.	apduZDP	-

ZDO Client

Mandatory. Processes ZDO client messages.

ZDO Client Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to ZDO client messages.	apduZDP	-

Device Annce Server

Mandatory. Processes device announcements.

Default Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to device announcement messages.	apduZDP	-

Active Ep Server

Mandatory. Processes active endpoint requests.

Active Ep Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to active endpoint request messages.	apduZDP	

Nwk Addr Server

Mandatory. Processes network address discovery requests.

Nwk Addr Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to net-work address discovery request messages.	apduZDP	

IEEE Address Server

Mandatory. Processes IEEE address discovery requests.

IEEE Address Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to IEEE address discovery request messages.	apduZDP	

System Server Discovery Server

Mandatory. Processes system server discovery requests.

System Server Discovery Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to system server discovery request messages.	apduZDP	

Permit Joining Server

Mandatory. Processes 'permit joining' requests.

Permit Joining Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to permit joining request messages.	apduZDP	

Node Descriptor Server

Mandatory. Processes Node descriptor requests.

Node Descriptor Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to node descriptor request messages.	apduZDP	

Power Descriptor Server

Mandatory. Processes Node Power descriptor requests.

Power Descriptor Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to power descriptor request messages.	apduZDP	

Match Descriptor Server

Mandatory. Processes Match descriptor requests.

Match Descriptor Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to Match descriptor request messages.	apduZDP	

Simple Descriptor Server

Mandatory. Processes simple descriptor requests.

Simple Descriptor Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to Simple descriptor request messages.	apduZDP	

Mgmt Lqi Server

Mandatory. Processes management LQI requests.

Mgmt Lqi Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to Link Quality Indicator (LQI) request messages.	apduZDP	

Mgmt Rtg Server

Mandatory. Processes management routing requests.

Mgmt Rtg Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to management routing request messages.	apduZDP	

Mgmt Leave Server

Mandatory. Processes management leave requests.

Mgmt Leave Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to management leave request messages.	apduZDP	

Mgmt NWK Update Server

Mandatory. Processes management network update requests.

Mgmt NWK Update Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to management network update request messages.	apduZDP	

Bind Unbind Server

Mandatory. Processes both bind and unbind requests.

Bind Unbind Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to bind and unbind request messages.	apduZDP	

Extended Active Ep Server

Mandatory. Processes extended active endpoint discovery requests.

Active Ep Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to extended active endpoint discovery request messages.	apduZDP	

Extended Simple Descriptor Server

Extended Simple Descriptor Server Parameters

Mandatory. Processes extended Simple descriptor discovery requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to extended Simple descriptor discovery request messages.	apduZDP	

End Device Bind Server

Mandatory (Coordinator only). Processes End Device bind requests.

End Device Bind Server Parameters

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to end device bind request messages.	apduZDP	
Timeout	Number of seconds before timing out an End Device bind request.	5	1 or higher
Bind Num Retries	Number of binding retries attempted if a binding request (zdo_bind_req or end_device_bind_req) fails.		

Parent Announcement Server

Mandatory on the coordinator and router devices.

End Parent Announcement Server

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to parent announce broadcast messages.		

Management Enhanced Network Update Server

Mandatory for Sub Gig; optional for 2.4G.

Management Enhanced Network Update Server

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to management enhanced network update request messages.	apduZDP	

MIB IEEE List Server

Mandatory on routers in Sub Gig; optional on 2.4G.

MIB IEEE List Server

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to MIB IEEE request messages.	apduZDP	

Parent topic:*ZigBee network parameters*

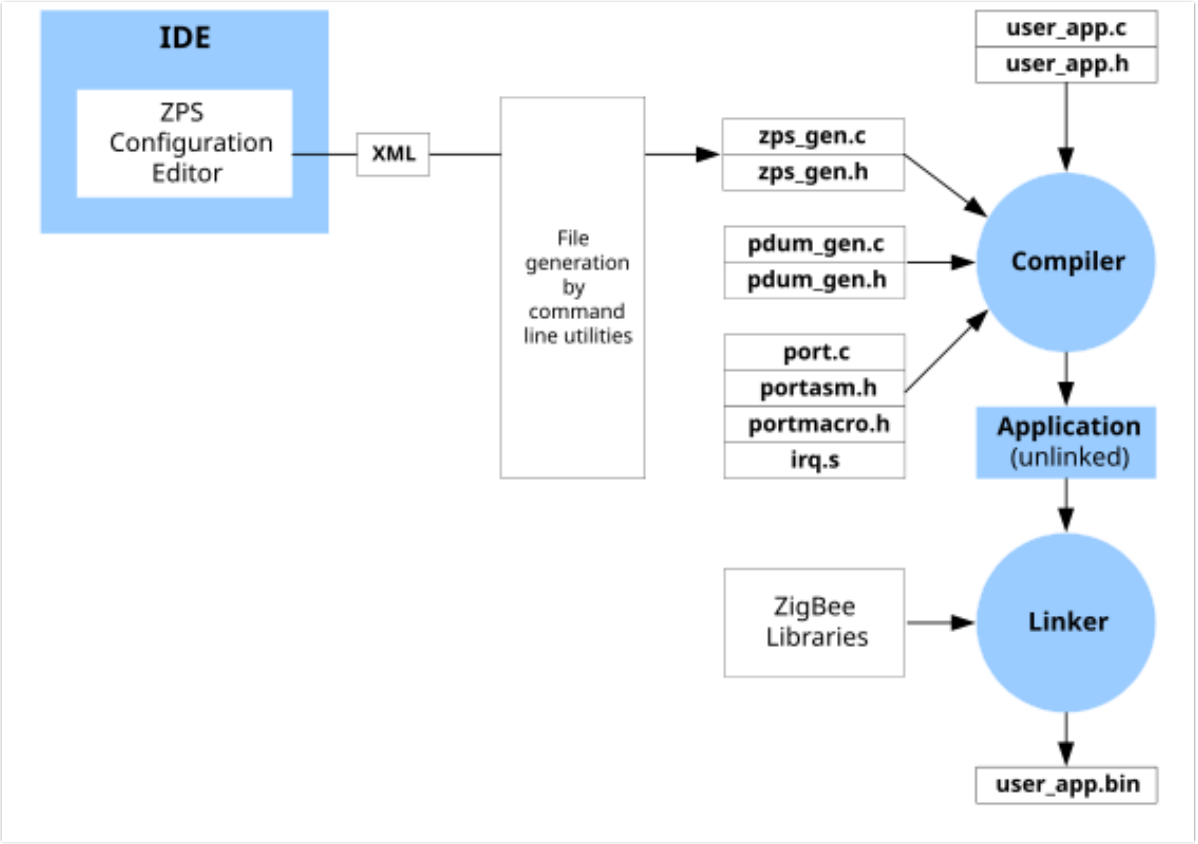
ZPS Configuration Editor In developing a ZigBee PRO application, certain static configuration is required before the application is built. The ZPS Configuration Editor is used to simplify this configuration. This editor is supplied as an NXP plug-in for Eclipse and is provided in the ZigBee 3.0 SDK (see Section 5.1.2 ZigBee 3.0 SDK). The plug-in is suitable for use with MCUXpresso software.

The ZPS Configuration Editor provides a convenient way to set ZigBee network parameters, such as the properties of the Coordinator, Routers and End Devices (for example, by setting elements of the device descriptors). This chapter describes how to use the ZPS Configuration Editor, as follows: Section 13.1

- Section 13.1 Configuration principles describes how the ZigBee network configuration is used in the application build process.
- Section 13.2 ZPS Configuration Editor wizard describes how to access the ZPS Configuration Editor wizard.
- Section 13.3 Overview of ZPS Configuration Editor Interface provides an overview of the interface provided by the ZPS Configuration Editor.
- Section 13.4 Using the ZPS Configuration Editor describes how to use the ZPS Configuration Editor to perform important configuration tasks.

Configuration principles The build process for a ZigBee PRO application takes a number of configuration files, in addition to the application source file and header file. The following files are generated from the MCUXpresso IDE to feed into the build process:

- ZigBee PRO Stack files:
  - zps_gen.c
  - zps_gen.h
- PDUM files:
  - pdum_gen.c
  - pdum_gen.h
- Other files:
  - port.c
  - portasm.h
  - portmacro
  - irq.s



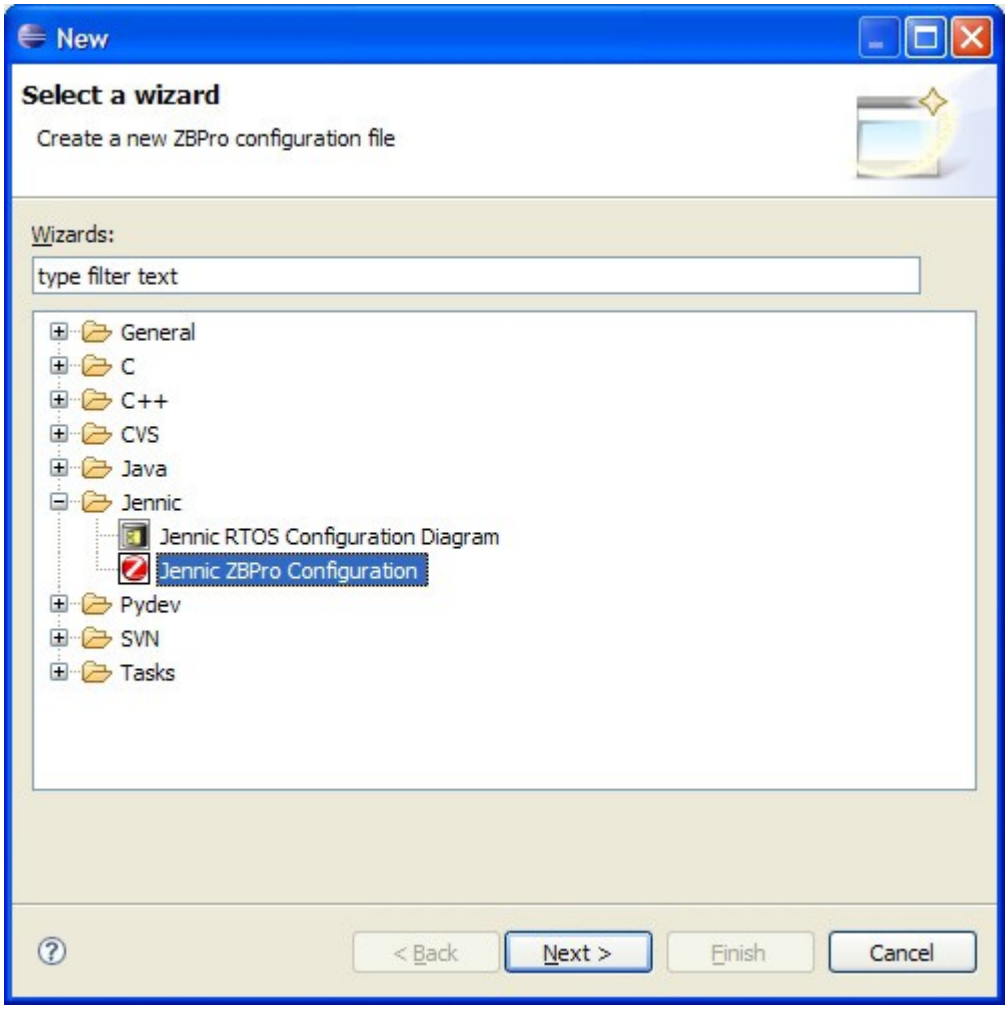
Parent topic:*ZPS Configuration Editor*

ZPS Configuration Editor wizard Before you can start to create a new ZigBee PRO stack configuration, the ZPS Configuration Editor plug-in must be installed in MCUXpresso.

To check if the plug-in is already installed, start MCUXpresso and select

File > **New** > **Other** from the main menu. Check that a **Jennic** option exists in the **Select a Wizard** dialogue box - expanding the **Jennic** option should show "ZBPro Configuration", as illustrated in the screenshot below.

Selecting a Wizard



Using the wizard highlighted in the screenshot above, you can start to create a new ZigBee PRO configuration, as described in Section 12.4.1.

If the wizard is not present, install the ZPS Configuration Editor plug-in, which is supplied in the ZigBee 3.0 SDK.

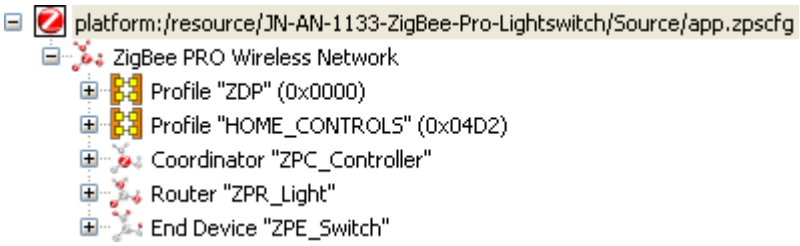
Parent topic:*ZPS Configuration Editor*

Overview of ZPS Configuration Editor Interface The ZPS Configuration Editor allows ZigBee network parameters to be configured through an easy-to-use Windows Explorer-style interface. This interface is outlined below.

The parameter values for the whole network are stored in a file with extension **.zpscfg**, and the ZPS Configuration Editor provides a convenient way to view and edit the contents of this file.

The network parameters are presented in an expandible tree, as shown below.

Network Parameters



The information under each of these entries is described below.

Entries that sit at the same level in the tree are termed 'siblings', while an entry that sits under another entry in the tree (a sub-entry) is termed a 'child'.

The top level of the tree shows the Extended PAN ID. The next level shows the following siblings:

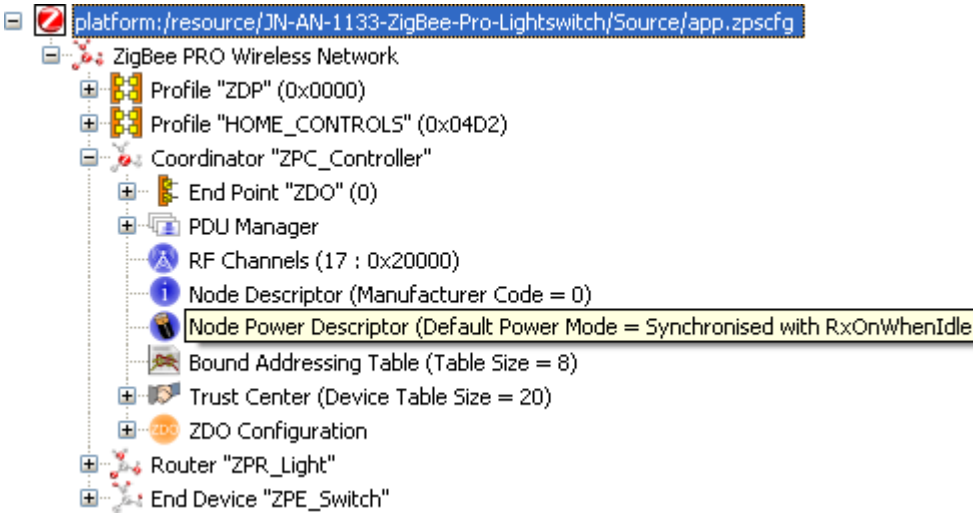
- Entries for the ZigBee application profiles used in the network
- Entry for the Coordinator
- Entries for the Routers
- Entries for the End Devices

Profile An application profile has a numeric ID and a name. The Profile entry contains child entries for the clusters supported by the profile - each cluster is identified by a numeric ID and a name.

Note: There must be entries for all application profiles supported by the network. An individual device may not use all profiles, although a device can use more than one profile to support multiple features (for example, measurement of temperature, humidity, and light level).

Parent topic: Overview of ZPS Configuration Editor Interface

Coordinator The Coordinator entry contains a name and a number of associated parameters, mainly related to the APS and NWK layers of the ZigBee PRO stack.



The child entries for the Coordinator are shown above and include the following:

- Endpoint entries, one for each endpoint on the Coordinator, with each endpoint having child entries specifying the input and output clusters used (note that each input cluster must be paired with an APDU).
- PDU Manager, with child entries specifying the APDUs used.
- Channel Mask, specifying the 2.4-GHz band channels to scan when creating the network.
- Node Descriptor for the Coordinator.
- Node Power Descriptor for the Coordinator.

Parent topic: Overview of ZPS Configuration Editor Interface

Router Each Router entry contains a name and a number of associated parameters, mainly related to the APS and NWK layers of the ZigBee PRO stack. The child entries for a Router include the following:

- Endpoint entries, one for each endpoint on the Router, with each endpoint having child entries specifying input and output clusters used (note that each input cluster must be paired with an APDU).
- PDU Manager, with child entries specifying the APDUs used.
- Channel Mask, specifying the 2.4-GHz band channels to scan when attempting to join a network.
- Node Descriptor for the Router.
- Node Power Descriptor for the Router.

Parent topic: Overview of ZPS Configuration Editor Interface

End Device Each End Device entry contains a name and a number of associated parameters, mainly related to the APS and NWK layers of the ZigBee PRO stack. The child entries for an End Device include the following:

- Endpoint entries, one for each endpoint on the End Device, with each endpoint having child entries specifying the input and output clusters used (note that each input cluster must be paired with an APDU).
- PDU Manager, with child entries specifying the APDUs used.
- Channel Mask, specifying the 2.4-GHz band channels to scan when attempting to join a network.
- Node Descriptor for the End Device.
- Node Power Descriptor for the End Device.

Parent topic: Overview of ZPS Configuration Editor Interface

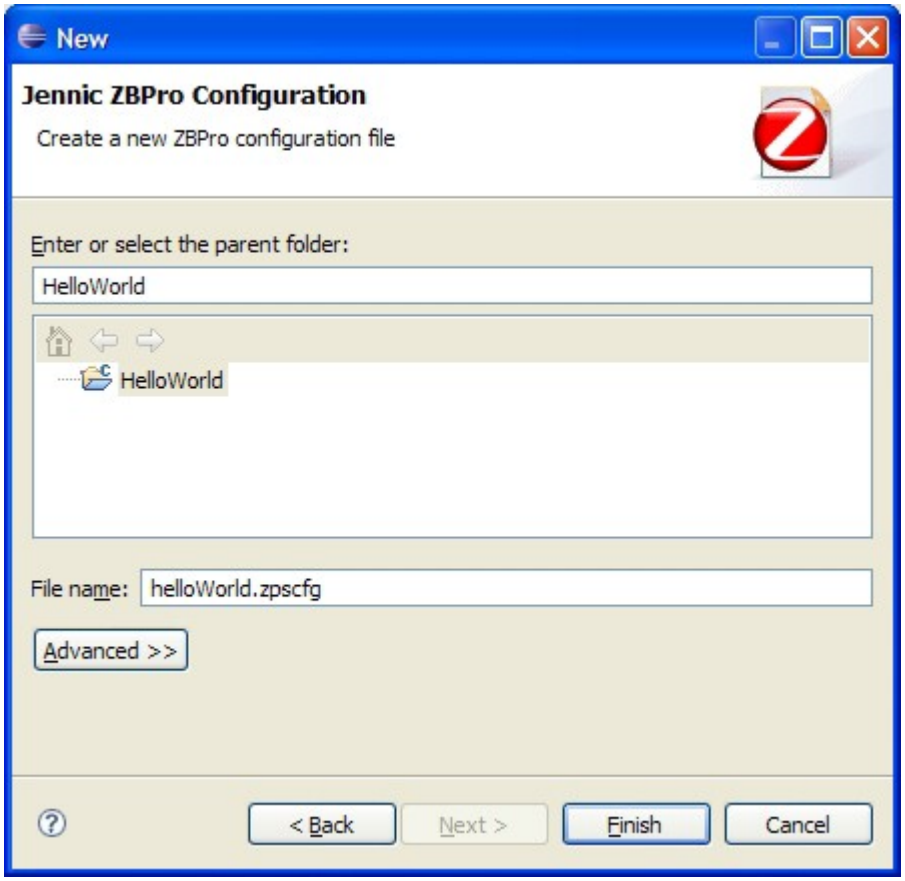
Parent topic:*ZPS Configuration Editor*

Using the ZPS Configuration Editor **Note:** This section assumes that you wish to add a ZigBee PRO stack configuration to a project which you have already created in MCUXpresso (in this example, HelloWorld).

Creating a New ZPS Configuration ***Step 1:*** In MCUXpresso, start the ZPS Configuration Editor wizard. To do this, follow the menu path **File** > **New** > **Other** and in the **Select a Wizard** dialogue box, select "Jennic ZBPro Configuration" and click **Next**(shown in Figure 13 on page 283).

The **New** dialogue box opens for the ZBPro Configuration.

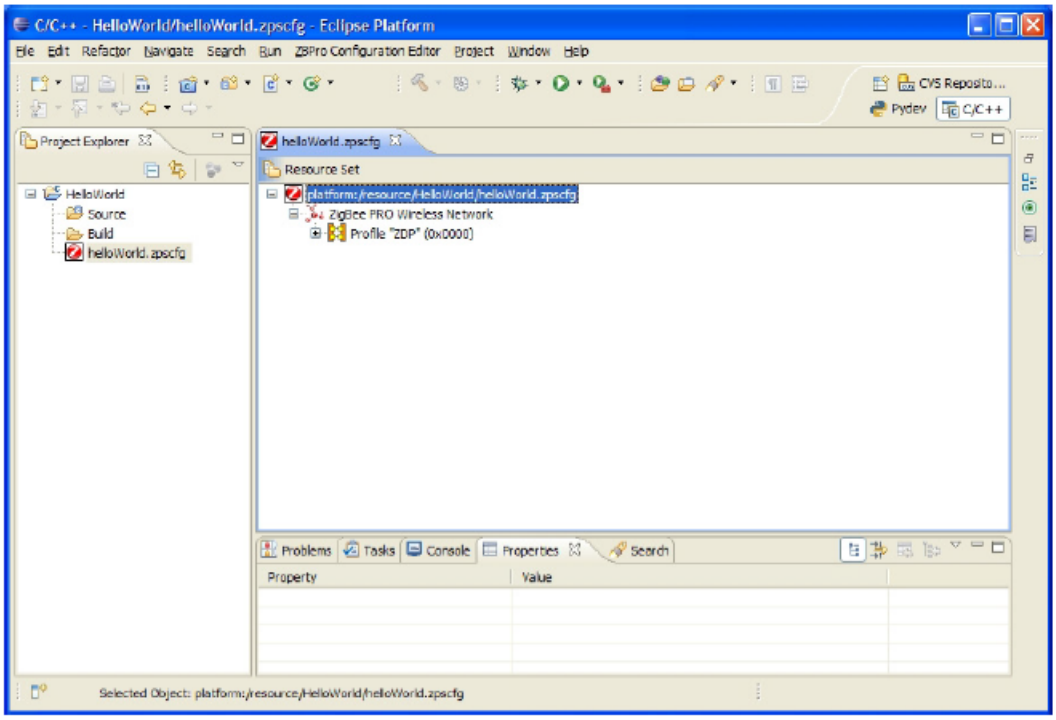
New ZPS Configuration



Step 2: Click on your project to select it as the parent folder. In the **File name**field, enter a name for the configuration file (keep the extension **.zpscfg**) and then click **Finish**.

A new configuration (with the default set of parameters) will open in the editor, as shown below.

ZPS Configuration Editor Window



Parent topic: Using the ZPS Configuration Editor

Adding Device Types Follow the steps below to add devices:

Step 1: Right-click on **ZigBee PRO Wireless Network**and select **New Child** > **Coordinator**from the drop-down menu. This inserts a Coordinator with the minimum necessary child elements.

Step 2: Add Routers and End Devices in the same way, as required. The network can only have one Coordinator, but as many different Router or End Device types (that is, running different application features and with different endpoints) as required.

Step 3: For each new device, use the **Properties**tab (bottom pane) to enter the required top-level parameters. For a sleeping End Device, set **Sleeping**to True (by right-clicking on the value and using the drop-down box).

Note: To display the advanced properties, click the Advanced tool button to the right of the Properties view tab. Refer to Section 13.4.4 Setting advanced device parameters. These properties are all set to default values and can be left unchanged, unless specific changes are required.

To add a profile ***Step 1*** Right-click on **ZigBee PRO Wireless Network** and select **New Child** > **Profile** from the drop-down menu. This inserts a profile with no child elements.

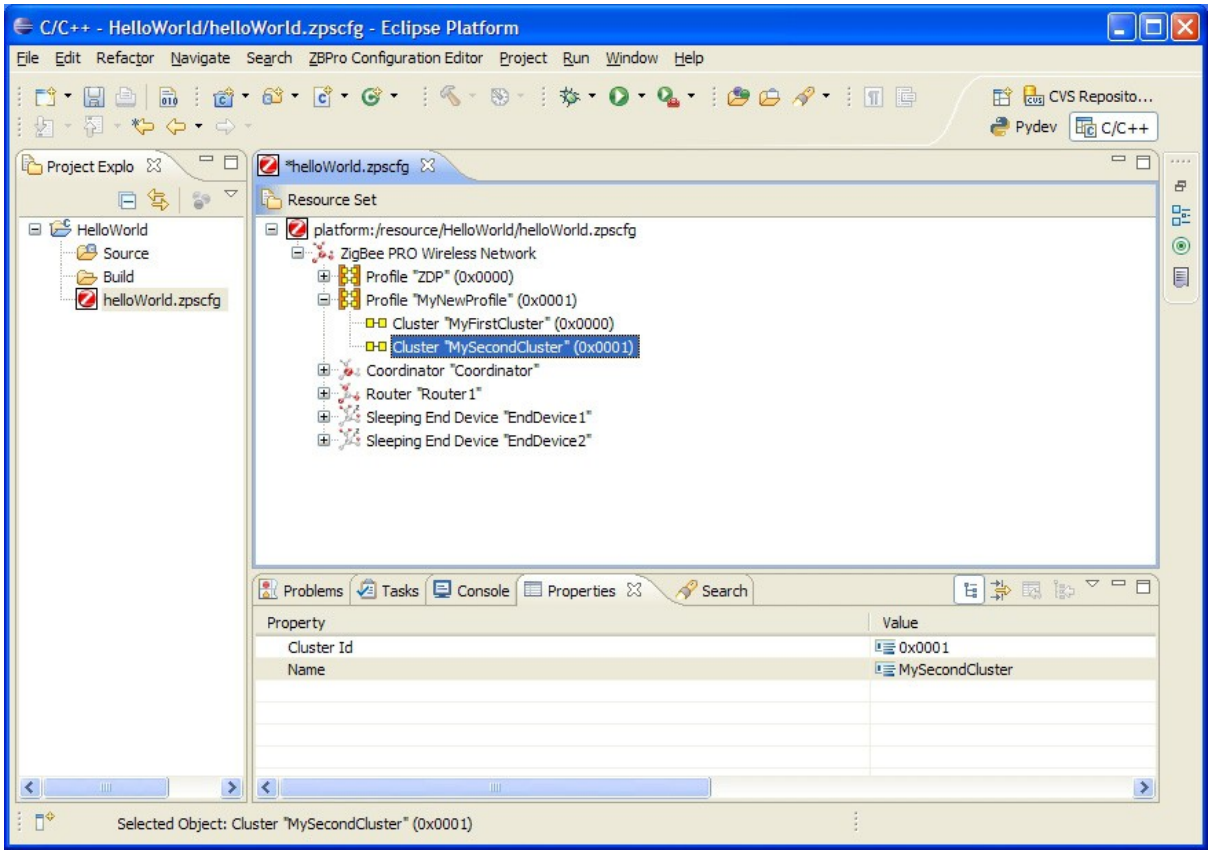
Step 2 Edit the properties in the **Properties** tab to set **Name** and **Id** for the new profile.

Parent topic: Adding Device Types

To add clusters to the new profile ***Step 1*** Right-click on the new profile created above and select **New Child** > **Cluster** from the drop-down menu.

Step 2 Edit the properties in the **Properties** tab to set **Name**and **Id** for the new cluster.

Step 3 Repeat Step 1 and Step 2 to add more clusters, as required.



Parent topic: Adding Device Types

Parent topic: Using the ZPS Configuration Editor

Setting Coordinator properties **To set the channel mask and Node Power descriptor, use the below steps:**

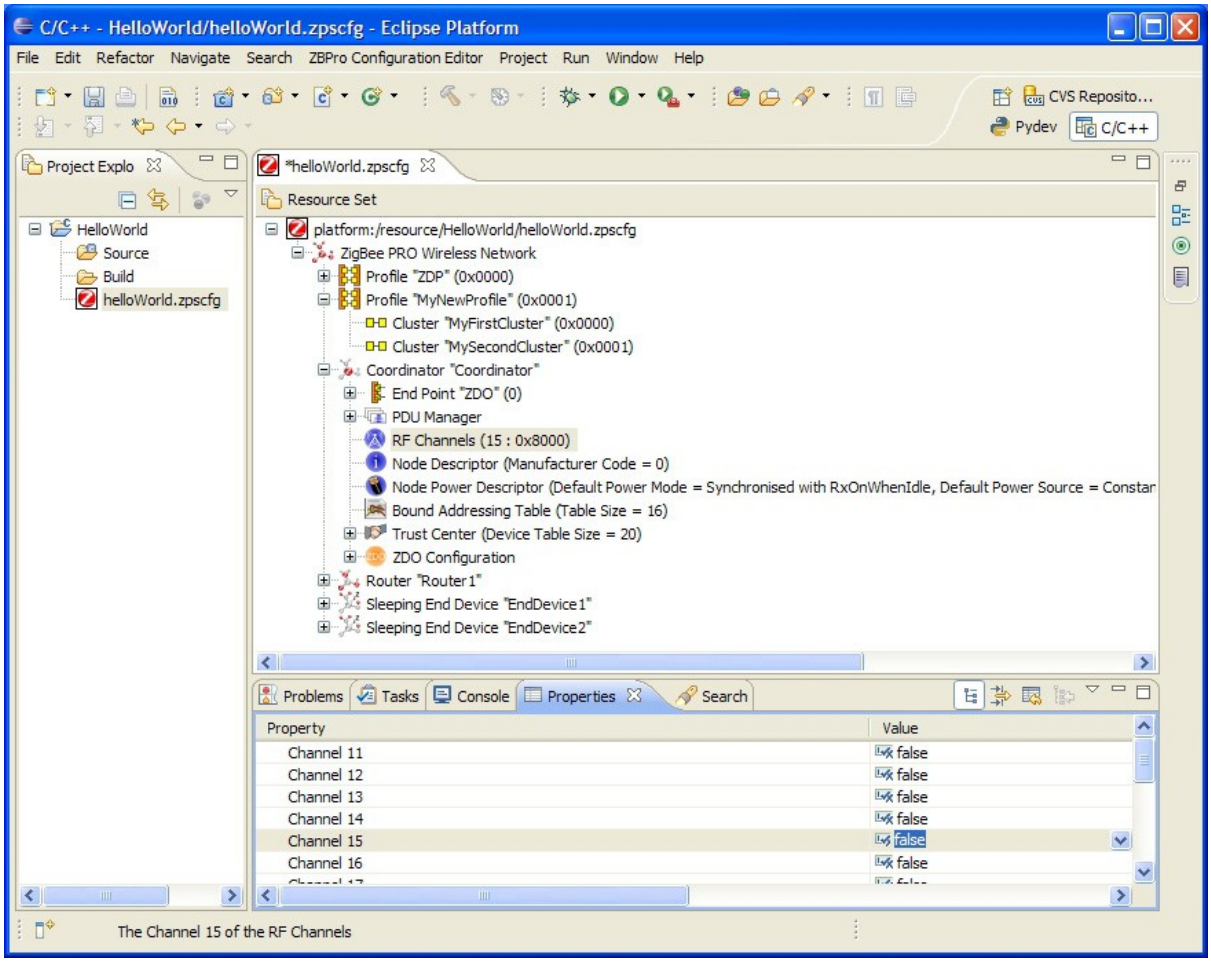
Step 1: Expand the Coordinator node in the editor. This reveals the default set of features for the Coordinator, ZDO endpoint, and ZDO servers.

Step 2: Click on the **RF Channels** element to modify the channel mask.

There are 16 channels available, numbered 11 to 26, which are now shown in the **Properties** tab. A single channel or a set of channels can be selected for the channel mask, as required.

Step 3: In the **Properties** tab, set the desired channel(s) to true (by right-clicking on the value and using the drop-down box).

Channel Mask Selection



Step 4: Click to select the **Node Power Descriptor**.

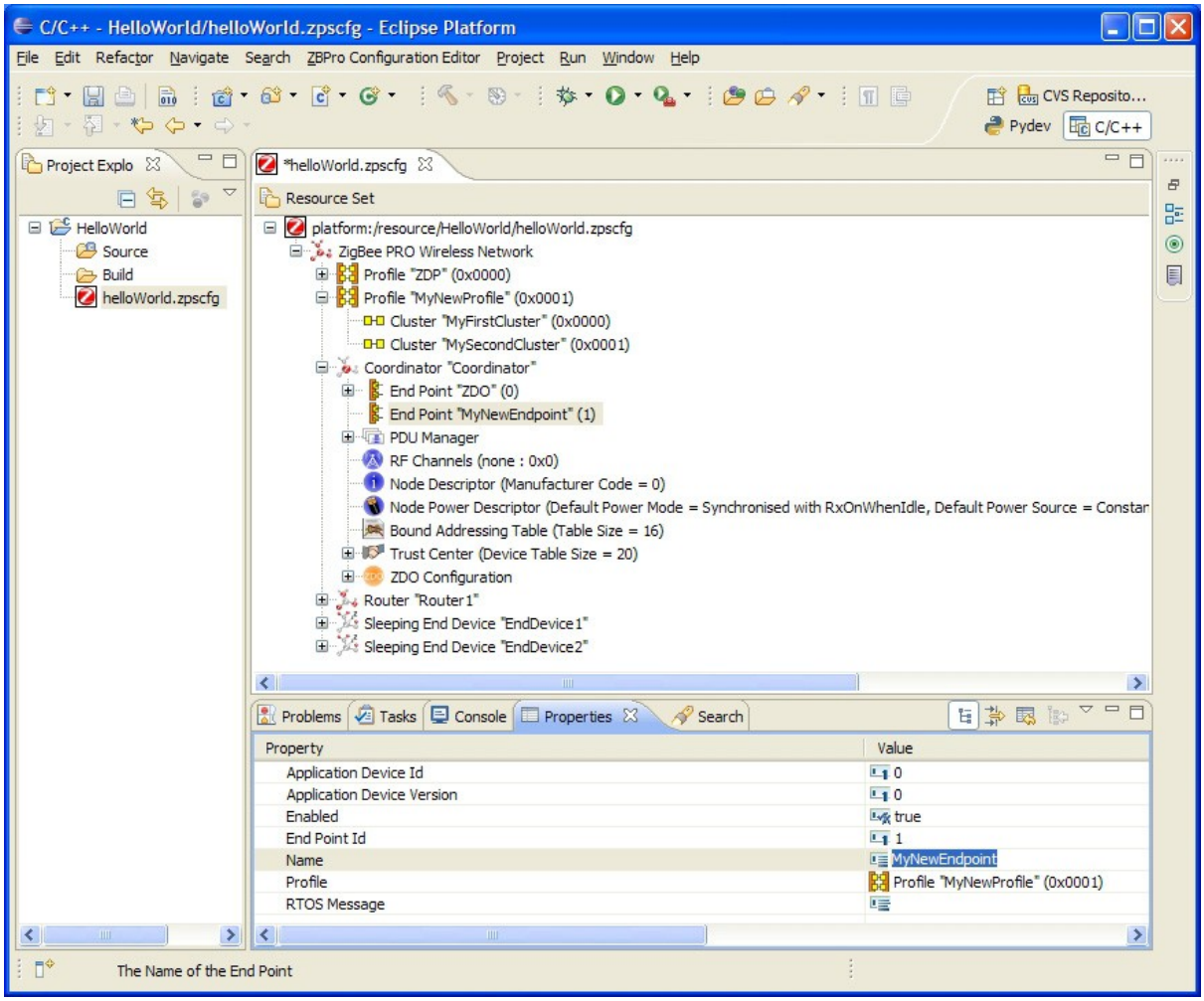
Step 5: Edit the properties in the **Properties** tab, as required.

To add a new endpoint ***Step 1:*** Right-click on the Coordinator node and select **New Child** > **End Point** from the drop-down menu.

Step 2: Edit the properties in the **Properties** tab to set **Name** and **Profile** for the endpoint (the profile is selected from the drop-down box).

Step 3: Repeat Steps 1 and 2 for as many endpoints as are required.

Endpoint Properties

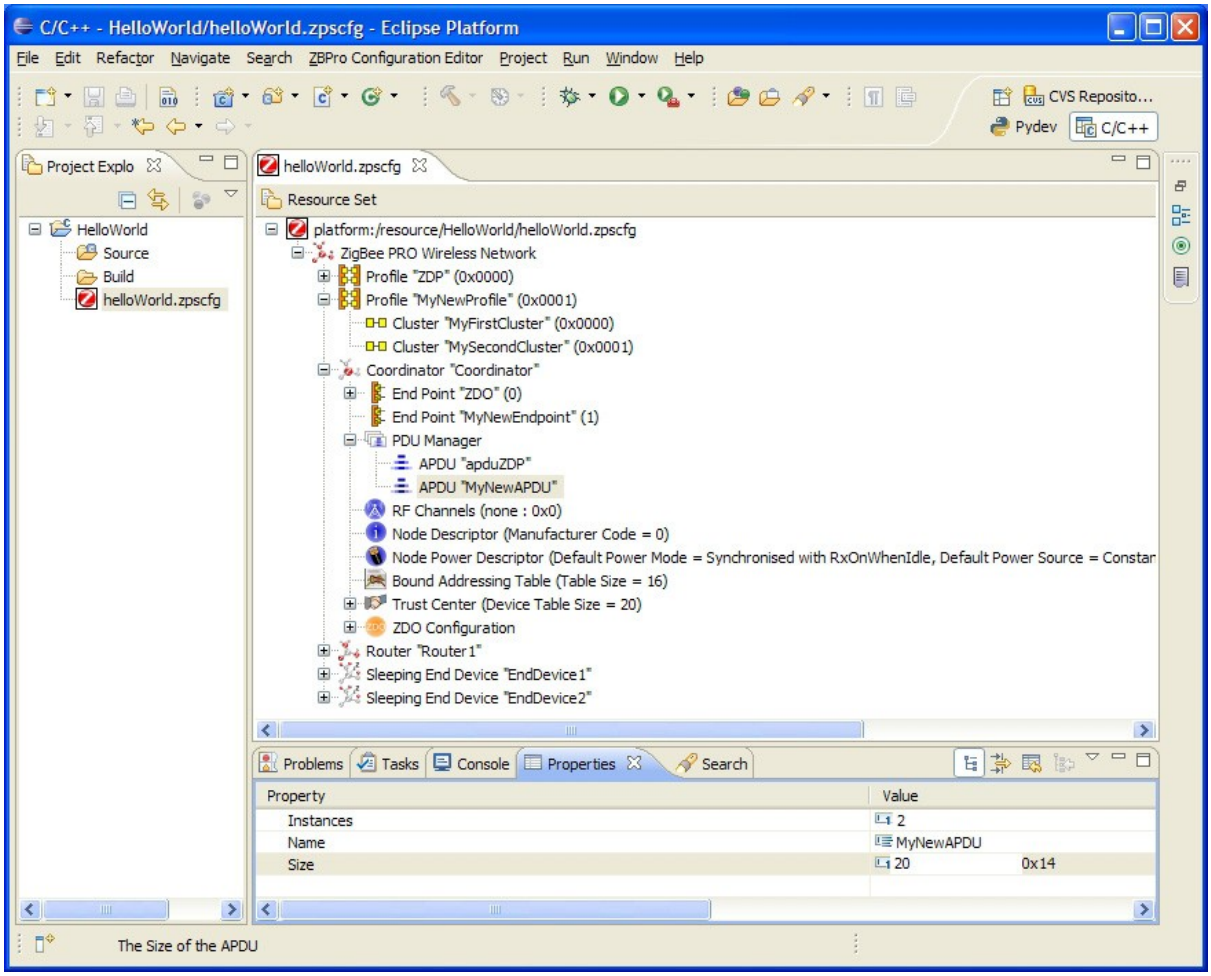


Parent topic: Setting Coordinator properties

To add an APDU At least one APDU is required before an endpoint can send or receive data. The same APDU can be used to send and receive data, or different APDUs can be set up for send and receive - this allows control of buffering and memory resources, and is the decision of the application designer.

Step 1: Right-click on **PDU Manager** and select **New Child** > **APDU** from the drop-down menu.

Step 2: Edit the properties in the **Properties** tab to set **Name**, **Instances** (number of) and **Size** (of each instance - this should be set to the size of the largest APDU to be received).



Parent topic: Setting Coordinator properties

To add input and output clusters to an endpoint

To add input and output clusters to an endpoint

Step 1: Right-click on the endpoint and select **New Child** > **Input Cluster**or **New Child** > **Output Cluster**, as required, from the drop-down menu.

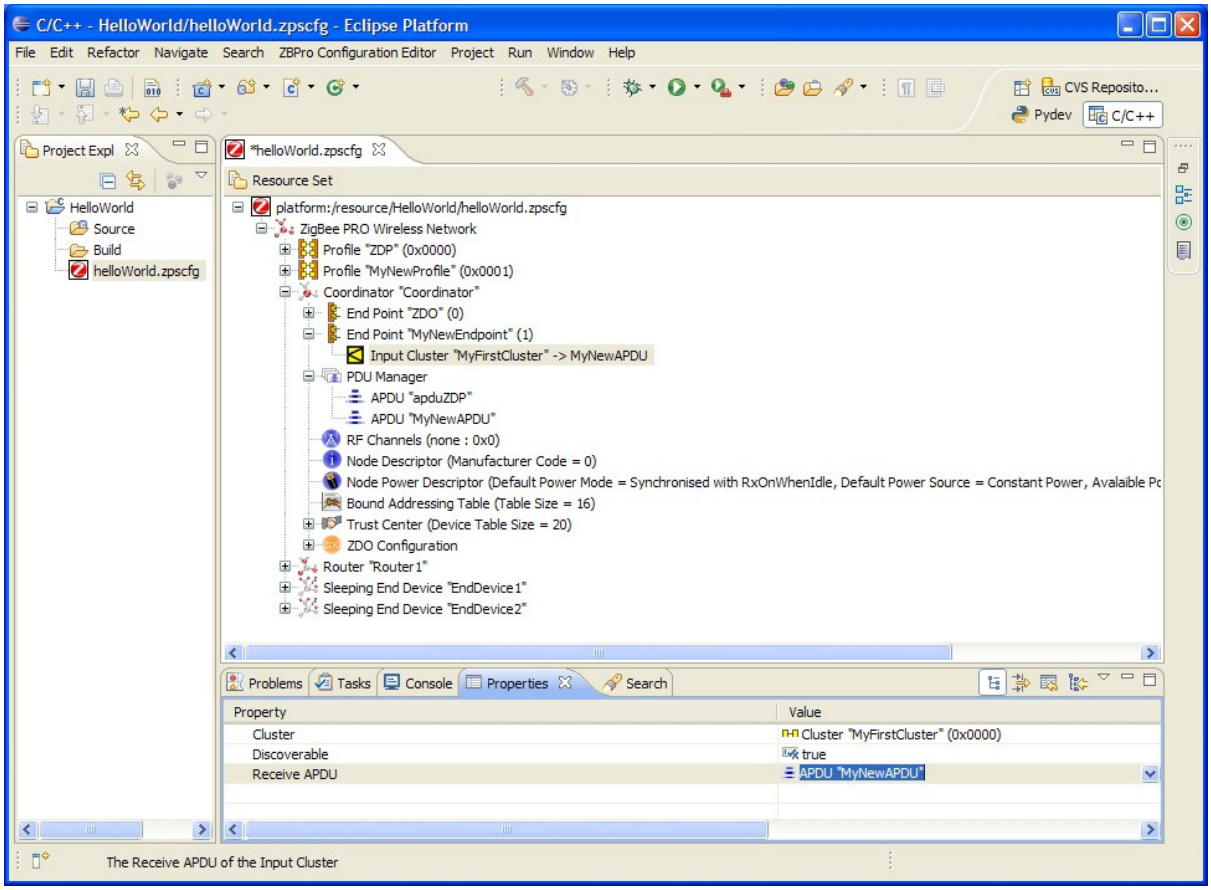
Step 2: Edit the properties in the **Properties**tab to set **Cluster**- select from the available clusters in the drop-down list.

Step 3: Edit the **Rx APDU**or **Tx APDU**property to assign an APDU to the cluster - select from the available APDUs in the drop-down list.

To receive data, a cluster must have an assigned APDU. The same cluster can be both an input and output cluster, i.e. it will both send and receive data.

When an endpoint with an output cluster sends data, the receiving endpoint must have an input cluster in order to receive the data, otherwise the stack will reject it and will not notify the receiving endpoint. However, the Default cluster can be added to the endpoint in order to deal with received data that is destined for input clusters not supported by the endpoint (see the Note below this procedure).

Step 4: Repeat Step 1 to Step 3 to add as many clusters as are required for the endpoint.



Step 5: Repeat Step 1 to Step 4 for Routers and End Devices, as required.

Note: In the above procedure, you may want to add the Default cluster (with a Cluster ID of 0xFFFF) as an input cluster. The inclusion of the Default cluster means that received messages that were intended for input clusters not supported by the endpoint will still be passed to the application. The messages must, however, come from defined application profiles, otherwise they are discarded.

Parent topic: Setting Coordinator properties

Parent topic: Using the ZPS Configuration Editor

Setting advanced device parameters You can set the advanced device parameters (detailed in Section 12.7) for a device as follows:

Step 1: Click on the relevant device (for example, Coordinator) in the **Resource Set** pane.

Step 2: Click on the **Advanced Device Parameters** button in the tool bar of the lower pane (indicated below).

Step 3: Edit the relevant parameters in the **Properties** tab of the lower pane.

Step 4: Save your settings.

The ZigBee PRO R22 version of the stack allows the presence of multiple MAC interfaces. This is to support both 2.4G and 868 MHz frequency bands using the single ZigBee stack. To address this, a MAC interface table needs to be configured in the ZPS Configuration diagram.

The MAC interface list can be found as an option for the node, for example, if you have ZigBee network with a router node. You can select the router node and press the right mouse button

to provide the options. The MAC interface list can be found under **New Child** > **Mac Interface List**.

After adding the MAC interface list, select and right-click on the MAC interface list to provide the options. The MAC interface can be found under **New Child** > **MAC Interface**.

After adding the MAC interface, the properties can be updated. The default is 2.4G. This default can be kept. The "**Router Allowed**" properties should be set to "**true**".

Note: Users should edit the advanced device parameters in order to change the Extended PAN ID (APS Use Extended PAN ID parameter) and the maximum number of children of the Coordinator or Router (Active Neighbor Table Size parameter) from the default values - see Section 6.1.1 and Section 6.1.2.

Parent topic: Using the ZPS Configuration Editor

Parent topic:*ZPS Configuration Editor*

Appendix A: Handling stack events The NXP ZigBee PRO stack events are listed below (they are detailed in Chapter 11.1):

```
ZPS_EVENT_NONE
ZPS_EVENT_APS_DATA_INDICATION
ZPS_EVENT_APS_DATA_CONFIRM
ZPS_EVENT_APS_DATA_ACK
ZPS_EVENT_NWK_STARTED
ZPS_EVENT_NWK_JOINED_AS_ROUTER
ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE
ZPS_EVENT_NWK_FAILED_TO_START
ZPS_EVENT_NWK_FAILED_TO_JOIN
ZPS_EVENT_NWK_NEW_NODE_HAS_JOINED
ZPS_EVENT_NWK_DISCOVERY_COMPLETE
ZPS_EVENT_NWK_LEAVE_INDICATION
ZPS_EVENT_NWK_LEAVE_CONFIRM
ZPS_EVENT_NWK_STATUS_INDICATION
ZPS_EVENT_NWK_ROUTE_DISCOVERY_CONFIRM
ZPS_EVENT_NWK_POLL_CONFIRM
ZPS_EVENT_NWK_ED_SCAN
ZPS_EVENT_ZDO_BIND
ZPS_EVENT_ZDO_UNBIND
ZPS_EVENT_ZDO_LINK_KEY
ZPS_EVENT_BIND_REQUEST_SERVER
ZPS_EVENT_ERROR
ZPS_EVENT_APS_INTERPAN_DATA_INDICATION
ZPS_EVENT_APS_INTERPAN_DATA_CONFIRM
ZPS_EVENT_TC_STATUS
```

These events are handled by the following stack-supplied callback function:

```
void APP_vGenCallback(uint8 u8Endpoint, ZPS_tsAfEvent *psStackEvent);
```

The stack populates psStackEvent with a pointer to one of the above events, when it occurs.

Note:

- The above callback function must be incorporated in your application code, otherwise your application will not compile.
- You are recommended to push the stack events onto a message queue for delayed processing rather than process the event in the callback function.

Appendix B: Application design notes This Chapter describes information and recommendations useful to designers who are incorporating non-routine operations in their applications.

The topics covered are:

- Fragmented data transfers
- Sending data to sleeping end devices
- Clearing stack context data before a rejoin
- Beacon filtering guidelines
- Table configuration guidelines
- Received message queues
- Noise threshold for forming a network

Fragmented data transfers The send 'with acknowledgment' functions (**ZPS_eAplAfUnicastAckDataReq()**, **ZPS_eAplAfUnicastIeeeAckDataReq()**, and **ZPS_eAplAfBoundAckDataReq()**) allow a large data packet to be sent that may be fragmented into multiple messages/ frames during transmission. As a general rule, one of these two functions should be used when sending a data packet with a payload size greater than 80 bytes. It is important to note, however, that the use of APS security will reduce this limit, as payload bytes are taken up by security data.

The processes of fragmentation at the sender and de-fragmentation at the receiver are transparent to the applications at the two ends, but the points described in the sub-sections below should be noted.

Note:

1. Fragmentation is described further in Fragmented data transmission to sleeping End Device: in connection with fragmented data transfers to sleeping End Devices.
2. The ZigBee network parameters referenced in this section are configured using the ZPS Configuration Editor and are described in *Chapter 13, "ZigBee Network Parameters"*. When setting up the APDUs to handle Rx fragmentation, care must be taken to ensure that the configuration setting in the ZPS Configuration Editor is sized to be able to handle 3* Tx Fragments.

Enabling/disabling fragmentation In order to allow fragmented data transfers between two nodes, you must appropriately configure two ZigBee network parameters:

- Set the parameter *Maximum Number of Transmitted Simultaneous Fragmented Messages*to a non-zero value on the sending node, to allow transmitted messages to be fragmented.
- Set the parameter *Maximum Number of Received Simultaneous Fragmented Messages*to a non-zero value on the receiving node, to allow received fragmented messages to be reassembled.

Note: Setting either of these parameters to zero would disable the corresponding fragmentation feature but reduce the size of your compiled application code.

Parent topic: Fragmented data transfers

Configuring acknowledgments You can configure how acknowledgments are generated during a fragmented data transfer by setting the ZigBee network parameter *APS Max Window Size*. This parameter must be set to the same value on the source and destination nodes. The parameter determines the number of fragments to be transferred before an acknowledgment is generated. For example, if a data packet is divided into 6 fragments and this parameter is set to 3, an acknowledgment will be generated after the third fragment and after the sixth fragment.

Note: Setting this parameter to a low value results in a high level of network traffic, since a large number of acknowledgment packets are sent.

The acknowledgment for a group of fragments contains an indication of any missing fragments from the group, thus requesting the missing fragment(s) to be re-sent.

Parent topic: Fragmented data transfers

Acknowledgment timeout A timeout of approximately 1600 ms is applied to each acknowledgment, measured from the time at which the last data fragment in the relevant group was transmitted - if no acknowledgment is received within this timeout period, the entire group of fragments is automatically re-sent. Up to 3 more re-tries can subsequently be performed. For a fragmented data transfer, the time that elapses before a completely unacknowledged transmission is abandoned is difficult to estimate, since this time depends on the number of fragments, the network parameter *APS Max Window Size*and the network parameter *APS Inter-frame Delay*(time between transmissions of consecutive fragments).

Parent topic: Fragmented data transfers

Parent topic:*Appendix B: Application design notes*

Sending data to sleeping end devices As described in Polling for Data, data sent to a sleeping End Device is buffered in the node's parent until the End Device collects the data through a polling mechanism, typically on waking from sleep. It is important that the polling interval is not too long, as the buffered data is discarded after 7 seconds. In addition, there is limited buffering space in the parent and the buffers are shared by all the children of the parent. Therefore, applications should be designed in such a way that data is only sent to a sleeping End Device when it is either awake or will wake in a timely manner to collect the data from its parent.

The following issues should also be considered when sending data to a sleeping End Device using one of the send '**with acknowledgment**' functions:

- ZPS_eAplAfUnicastAckDataReq()
- ZPS_eAplAfUnicastIeeeAckDataReq()
- ZPS_eAplAfBoundAckDataReq()

Note: The ZigBee network parameters referenced in this appendix are configured using the steps described in *Chapter 13, ZPS Configuration Editor*.

Acknowledged data transmission to sleeping end device When data is sent and an acknowledgment is required from the receiver, a timeout of approximately 1600 ms is applied to the acknowledgment. If no acknowledgment is received by the sender within this timeout period, the data is automatically re-sent. Up to 3 more re-tries can subsequently be performed, totalling just over 3 seconds before the data transfer is finally abandoned.

In the case of data sent to a sleeping End Device, the acknowledgment is generated by the End Device after collecting the data from its parent. Thus, if the data is not collected within the acknowledgment timeout period, the data is re-sent to the End Device (via its parent).

Note: There can be a case when the buffered data is collected by the End Device after the final re-try by the sender but before the data is discarded by the parent (between approximately 3 and 7 seconds after the initial transmission). In such cases, the acknowledgment that is eventually generated by the End Device is ignored by the sender, since the transaction has already timed out and terminated.

Parent topic: Sending data to sleeping end devices

Fragmented data transmission to sleeping End Device The Section 6.5.1 and Fragmented data transfers explain how the send 'with acknowledgment' functions can be used to send large

data packets that require to be fragmented into multiple NPDUs during transmission. Therefore, when sending a fragmented data packet to a sleeping End Device, the issues described in Acknowledged data transmission to sleeping end device apply.

In such a data transfer, the End Device should aim to collect all buffered data fragments from its parent before the transfer has completely timed out on the sender. Once the sender has abandoned the transaction, it does not respond to any acknowledgments requesting missing fragments (see Appendix B.1).

Once the End Device starts to receive fragmented data, it stays awake until the transaction is complete and runs its own poll timer to automatically collect each fragment - the polling period for this timer is set through the ZigBee advanced device parameter *APS Poll Period*. This poll timer runs for the duration of the fragmented transaction and then stops. The responsibility for polling then returns to the application.

Sending fragmented data to a sleeping End Device is likely to result in duplicate fragments of the message being sent. A list of the last few fragments received, called the APS Duplicate table, is maintained in the End Device. This table allows new fragments to be compared with previous fragments and duplicates identified. The maximum number of entries (fragments) in this table can be configured through the network parameter *APS Duplicate Table Size*. This table size should not be made too small, as a short table prevents duplicate fragments from being caught (4 may be a suitable value). This value should be considered in conjunction with the value of the network parameter *APS Persistence Time*. This parameter represents the time for which resources associated with a message are retained after the complete message has been received. Once the resources have been released, they may be used for a new transaction) - during this period, any duplicate fragments that are received are ignored.

Parent topic: Sending data to sleeping end devices

Parent topic:*Appendix B: Application design notes*

Clearing stack context data before a rejoin If a node rejoins the same secured network (with ZigBee PRO security enabled) but its stack context data was cleared before the rejoin (by calling ***NvErase*()**), data sent by the node will be rejected by the destination node since the frame counter has been reset on the source node - frame counters are described in Section 2.8 and *Appendix A*.

- Sent data will be accepted again by the destination node when the frame counter for the source node reaches its last count known before the rejoin. Therefore, you are not recommended to clear the stack context data before a rejoin.

However, it is worth noting that frame counters are reset across the entire network when a new network key is broadcast by the Trust Centre using the function **ZPS_eAplZdoTransportNwkKey()** - see Section 6.8.4. Thus, if stack context data is cleared before a rejoin, the frame counter problem can be avoided by broadcasting a new network key from the Trust Centre (normally the Coordinator) immediately after the rejoin.

To restore the stack to a default state and not clear the frame counters ZPS_vDefaultStack should be used - see Section 7.1.1.

Parent topic:*Appendix B: Application design notes*

Beacon filtering guidelines A filter can be introduced for filtering beacons in network searches (on a Router or End Device). Beacons can be filtered on the basis of Extended PAN ID (EPID), LQI value and device joining status/capacity (see below). The filter can be applied using the function **ZPS_bAppAddBeaconFilter()**.

If required, the above function must be called immediately before **ZPS_eAplZdoDiscoverNetworks()**, **ZPS_eAplZdoRejoinNetwork()** or **ZPS_eAplZdoStartStack()**.

A tsBeaconFilterTypestructure is supplied to the **ZPS_bAppAddBeaconFilter()**

function in order to specify the details of the filter to be implemented, including:

- A blacklist or whitelist of networks in terms of a list of EPIDs.
- The PAN ID of the network from which acceptable beacons should come.
- The minimum LQI value of an acceptable beacon.
- Flags indicating the properties on which beacons will be filtered, which include:
  - LQI value of beacon.
  - Permit Join enabled on sending device.
  - Capacity of sending device to accept Router children.
  - Capacity of sending device to accept End Device children.

After each discovery or rejoin, the flags are cleared while all other fields of the structure remain intact. The structure is detailed in Section 8.2.3.5.

The following general guidelines should be followed in using beacon filters:

- **Do not** implement a filter unless attempting a join, as this would prevent some stack operations from working correctly.
- **Do not** enable a blacklist and whitelist at the same time.
- **Do not** declare your filter structure as a local variable in a function, as it needs to exist for the duration of the discovery.

The following guidelines are relevant to network rejoins and associations.

Network rejoin

- **Do** set up a whitelist containing a single EPID corresponding to the network that the node is to rejoin (if only one network is of interest) and/or the PAN ID of this network.
- **Do** set up an LQI filter to reject distant beacons, if required.
- **Do not** enable filtering on Permit Join or Router/End Device Capacity.

Parent topic: Beacon filtering guidelines

Association

- **Do** set up an LQI filter to reject distant beacons, if required.
- **Do** filter on the Permit Join status to only find potential parents and networks that are accepting association requests.
- **Do** filter on Router/End Device Capacity, if required, depending on device type.

Note: A blacklist can be built up over several attempts to discover and associate, by keeping on adding to the array of EPIDs, as each network is rejected.

Parent topic: Beacon filtering guidelines

Parent topic:*Appendix B: Application design notes*

Table configuration guidelines This section provides guidelines on configuring various tables used by the ZigBee PRO stack. These tables can be configured through ZigBee network parameters in the ZPS Configuration Editor. The tables are sized, by default, to support a network of up to 250 nodes. The table sizes can be increased to support more nodes, but this will be at the expense of RAM and/or Flash usage.

The tables and their configuration are individually described in the sections below, which reference to the ZigBee network parameters used to configure the table sizes (the network parameters are detailed in *Chapter 12*).

Neighbor table The Neighbor table on a routing node (Router or Coordinator) holds information about the node's immediate neighbors:

- The first entry in the table contains information about the node's parent.
- Part of the table holds information about child nodes which have joined the network through the local device.
- The rest of the table holds information about nodes which are neither children nor the parent (these 'other' nodes are only relevant to Mesh networks).

The Neighbor table size is, by default, set to 26 - this is the minimum size required for a ZigBee-Compliant Platform. The table size may be increased through the parameter *Active Neighbor Table Size*to reflect the density of the network. However, increasing the table size uses more RAM. Increasing the Neighbor table size beyond 26 also results in an extra link status packet since one of these packets can accommodate a maximum of 26 neighbors. Thus it doubles the traffic for these periodic packets.

The first two parts of the Neighbor table, for the device's parent and children, form a sub-table that is persisted in Flash. This sub-table must not occupy more than two-thirds of the Neighbor table. Since this sub-table contains child entries, the size of the sub-table determines the number of children that the device is allowed to have - the maximum number of children is one less than the sub-table size.

The default size for the sub-table is 5, allowing up to 4 child nodes, but the size can be changed through the parameter *Child Table Size* (which corresponds to the total number of sub-table entries including the parent's entry, not just the child entries).

Note: Increasing the sub-table size uses more Flash for persisted data.

Parent topic: Table configuration guidelines

Address Map table The Address Map table on a node is used to keep a record of the address-pairs of network nodes with which the local node needs to communicate directly - that is, the IEEE/MAC address and network address of each of these nodes. In fact, an Address Map table entry only contains an index to an entry in the MAC Address table, where the actual addresses of the node are stored (see MAC Address table). The population of these tables is done as the result of device announcement messages.

The default size of the Address Map table is 10, but the size can be changed through the parameter *Address Map Table Size*. The Address Map table is fully persisted in Flash. Therefore, increasing the size of this table will impact both RAM and Flash usage.

Parent topic: Table configuration guidelines

MAC Address table The MAC Address table on a node is used to store the address-pairs of other network nodes - that is, the IEEE/MAC address and network address of each of these nodes. The entries in the MAC Address table are referenced from entries of both the Neighbor table and Address Map table. Therefore, the MAC Address table should be sized according to the combined sizes of the Neighbor table and Address Map table.

The default size of the MAC Address table is 36, but the size can be changed through the parameter *Maximum Number of Nodes*. The MAC Address table is fully persisted in Flash. Therefore, increasing the size of this table impacts both RAM and Flash usage.

Parent topic: Table configuration guidelines

Routing table A Routing table is held by the Coordinator and Router nodes to store routing information to other nodes in the network.

The default size of the Routing table is 70, which should be sufficient for most applications, but the size can be changed through the parameter *Routing Table Size*. The table size should be increased if routing bottlenecks are observed. The Coordinator needs to store routes to all the nodes in the network if it is required to communicate with every node. In this case, the Routing table size should be increased to the size of the network.

The Routing table is not persisted. Therefore, any increase only affects the RAM usage.

Parent topic: Table configuration guidelines

Broadcast Transaction table The Broadcast Transaction table is used for the origination, processing and passive acknowledgment of broadcast transmissions. The minimum required size of this table for a ZigBee-Compliant Platform is 9. However, an application that produces a large number of broadcasts may need a larger table. The size of the table can be set through the parameter *Broadcast Transaction Table Size*.

Parent topic: Table configuration guidelines

Route Discovery table The Route Discovery table is used to hold temporary details of a route discovery transaction. The table size dictates how many individual route discoveries can occur on the local node at a given time. The default size of the Route Discovery table is 2, but the size can be changed through the parameter *Route Discovery Table Size*. The default value severely restricts the number route discoveries and hence broadcasts on the network. Increasing the table size also requires increases in the Routing table and Broadcast Transaction table sizes.

The Route Discovery table is not persisted. Therefore, any increase only affects the RAM usage.

Parent topic: Table configuration guidelines

Discovery table A Discovery table is held by the Router and End Device nodes to store the results of a channel scan when searching for a network to join. The default size of the Discovery table is 8, but the size can be changed through the parameter *Discovery Neighbor Table Size*.

Parent topic: Table configuration guidelines

Route Record table The Route Record table is only relevant to a device, which will be the concentrator in a network, if many-to-one routing is implemented. This table replaces the Routing table in the node.

- The size of the Route Record table can be set through the parameter *Route Record Table Size*.
- In the concentrator node, this table size should be set to the size of the network. Since this table then replaces the Routing table in the node, the Routing table size should be set to 1 (see Appendix B.5.4).
- In all other network nodes, the size of the Route Record table should be set to 1.

Parent topic: Table configuration guidelines

Parent topic:*Appendix B: Application design notes*

Received message queues All messages received ZPS_msgMcpsInd on a network node are pushed into one of the following two queues:

- ZPS_msgMcpsDcfm

- ZPS_msgMcpsInd

These queues must be created by the application using the function **ZQ_vZQueueCreate()**. An example code is described in Section 6.9.1.2.

ZPS_msgMcpsDcfm All IEEE 802.15.4 MAC data deferred confirm events are added to this queue. The default size of this queue is 8 but a different queue size can be set when the queue is created. The queue can overflow if there is heavy network traffic.

Parent topic: Received message queues

ZPS_msgMcpsInd All IEEE 802.15.4 MAC data packets are added to this queue. The default size of this queue is 24 but a different queue size can be set when the queue is created. The queue can overflow if there is heavy network traffic.

Parent topic: Received message queues

Parent topic:*Appendix B: Application design notes*

Noise threshold for forming a network The ZigBee PRO stack provides a mechanism for forming a new network in the quietest IEEE802.15.4 radio channel. The Coordinator (centralized network) or Router (distributed network) that forms the network performs a channel scan to listen for activity from other local networks.

During the channel scan, the activity in each channel is assigned a noise level in the range 0 to 254. This result is compared with a noise level threshold, which is defined by the NIB value u8VsFormEdThreshold (which is part of the structure ZPS_tsNwkNibInitialValues). If the measured noise level for a channel is above this threshold, the channel is excluded from further consideration. Therefore, if all the channels in the scan are noisier than the threshold allows, no network is formed.

The stack then re-scans the channels that passed the noise threshold test (if any) and selects the one with the lowest beacon count in which to form the network.

Note:

- This assessment takes into account IEEE802.15.4 beacons only and no activity from networks based on other systems, such as Wi-Fi.
- The assessment is based on beacons only and does not consider the noise levels of the short-listed channels.

Default Behavior To avoid the situation in which no network is formed, the default value of u8VsFormEdThresholdis 0xFF, which is a special value and not a noise threshold. In this case, the network is always formed in the channel with the lowest IEEE802.15.4 beacon activity (no noise level assessment is performed).

Parent topic: Noise threshold for forming a network

Customizing the scan You can implement network formation based on the noise level threshold, as described above, by setting u8VsFormEdThresholdto an appropriate value in the range 0 to 254. In the following code fragment, a noise level threshold of 100 is set:

```
ZPS_psNwkNibGetHandle(ZPS_pvAplZdoGetNwkHandle())->u8VsFormEdThreshold = 100
```

Thus, in the above case, all channels with a noise level above 100 will be rejected.

If no suitable channel is found and no network formed, the application can dynamically increase the value of u8VsFormEdThresholdand initiate another scan.

Parent topic: Noise threshold for forming a network

Parent topic:*Appendix B: Application design notes*

Appendix C: Implementation of frame counters Frame counters are used to prevent message replay attacks (see Section 2.10, "Secure operating environment"). Each message sent across the network carries a frame count. It is inserted by the sending node and is part of a sequence of frame counter values for messages from this node.

Two types of frame counter exist:

- **NWK frame counters:** These are implemented automatically at the NWK layer of the stack and are used in all network communications.
- **APS frame counters:** These are optional and applied by the APS layer of the stack to one or more specific source/destination links.

The rest of this section describes the standard NWK frame counters. Detailed information on NWK and APS frame counters can be found in the ZigBee 3.0 specification.

A node locally maintains two types of NWK frame counters:

- **Outgoing frame counter:** A single frame counter for all sent messages is maintained in RAM. It is incremented every time a message is transmitted (to any destination node) and its value is inserted in the message. Its value can be persisted in non-volatile memory - the NXP method of persisting this counter is described later.
- **Incoming frame counters:** A frame counter is maintained in RAM for every node in the local Neighbor table. It records the frame count contained in the last message received from this node. Persisting the frame counters for all neighbors would use too much space in non-volatile memory. Hence, its value is not persisted. On a power-cycle or reset, these values are all reset to 0.

If a destination node receives a message containing a frame count which is less than the locally held incoming frame counter value for the same source node (corresponding to the frame count of the previous message received from the source node), it rejects the message. The receiving node informs the rest of the network by sending out a Network Status command with a status value of 0x11 (bad frame counter).

A problem might occur if the outgoing frame counter is not persisted and reverts to 0 when the node is reset. This would result in messages being sent with a frame count lower than the value expected by the destination node(s). Consequently, the messages from this node would be rejected.

Persisting the outgoing frame counter is therefore advisable, but doing this on every update of the counter causes excessive Flash wear. To minimize this wear, the NXP stack software only persists the outgoing frame counter every time it reaches a multiple of 1024. If the node is reset, the current frame counter value is likely to be greater than the persisted value, so 1024 is added to the recovered value. This ensures that the outgoing frame counter is always larger than the value in the last message sent before the reset.

For example, consider a case when the last message transmitted had a frame count of 4000 and then the node is reset. In such a case, the last persisted value of 3072 is recovered but 1024 is added to it to make an outgoing frame counter of 4096 (which will be persisted).

Appendix D: storing applications in device flash memory During start-up, the device bootloader (provided in internal ROM) searches for a valid application image in internal Flash memory. If it is present, then the device boots directly from Flash memory. If no image is found, the bootloader drops into In System Programming (ISP) mode.

Appendix E: Glossary of terms **Terms and their description**

Term	Description
Address	A numeric value that is used to identify a network node. In ZigBee, the device's
AIB	APS Information Base: A database for the Application Support (APS) layer of th
APDU	Application Protocol Data Unit: Part of a wireless network message that is han
API	Application Programming Interface: A set of programming functions that can l
APS	Application Support: A sub-layer of the Application layer of the ZigBee stack, r
Application	The program that deals with the input/output/processing requirements of the n
Application Profile	A collection of device descriptors that characterize an application for a particu
Attribute	A data entity used by an application, for example, a temperature measurement
Binding	The process of associating an endpoint on one node with an endpoint on anoth
Channel	A narrow frequency range within the designated radio band - for example, the
Child	A node which is connected directly to a parent node and for which the parent n
Cluster	A collection of attributes and commands associated with the endpoint for an a
Context Data	Data which reflects the current state of the node. The context data must be pre
Coordinator	The node through which a network is started, initialized and formed. The Coor
End Device	A node which has no networking role (such as routing) and is only concerned w
Endpoint	A software entity that acts as a communications port for an application on a Zi
Extended PAN ID (EPID)	A 64-bit identifier for a ZigBee PRO network that is assigned when the net- wor
IEEE 802.15.4	A standard network protocol that is used as the lowest level of the ZigBee softw
IEEE/MAC Address	A unique 64-bit address that is allocated to a device at the time of manufacture
Joining	The process by which a device becomes a node of a network. The device transm
Mesh Network	A wireless network topology in which all routing nodes (Routers and the Coord
Network Address	A 16-bit address that is allocated to a ZigBee node when it joins a network. The
NIB	NWK Information Base: A database containing attributes needed in the manag
Node Descriptor	A set of information about the capabilities of a node.
Node Power Descriptor	A set of information about a node's current and potential power supply.
NPDU	Network Protocol Data Unit: The transmitted form of a wireless network messa
PAN ID	Personal Area Network Identifier: This is a 16-bit value that uniquely identifies
Parent	A node which allows other nodes (children) to join the network through it and
Router	A node which provides routing functionality (in addition to input/output/proce
Routing	The ability of a node to pass messages from one node to another, acting as a ste
Simple Descriptor	A set of assorted information about a particular application/endpoint.
Sleep Mode	An operating state of a node in which the device consumes minimal power. Du
Stack	The hierarchical set of software layers used to operate a system. The high- leve
Stack Profile	The set of features implemented from the ZigBee specification - that is, all the n
UART	Universal Asynchronous Receiver Transmitter: A standard interface used for c
User Descriptor	A user-defined description of a node (for example, "KitchenLight").
ZigBee Base Device	A framework for the use of ZigBee device types that provides basic functionali
ZigBee Certified Product	An end-product that uses ZigBee Compliant Platforms and public Application P
ZigBee Cluster Library (ZCL)	A collection of clusters that can be individually employed in ZigBee devices, as
ZigBee Compliant Platform	A component (such as a module) that has been tested for ZigBee compliance an
ZigBee Device Objects (ZDO)	A special application which resides in the Application Layer on all nodes and p

Revision history *Table 1* summarizes the changes to this document.

Document ID	Release Date	Description
JNUG3130_v	10 March 2025	Added support for RW612 devices
JNUG3130_v	24 January 2025	Added support for MCXW71 and MCXW72 devices
JNUG3130_v	2 March 2023	Added support for K32W1 devices
JNUG3130_v	29 September 2022	Updated the document template. Other minor updates
JNUG3130_v	6 December 2021	- Added new values in Section 11.2.5, “Extended error codes”.Updated the document template.
JNUG3130_v	18 November 2019	Updated for K32W\JN5189
JNUG3130_v	19 June 2018	First release

Note about the source code in the document Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2018-2025 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

ZigBee 3.0 Devices User Guide

Overview This manual describes ZigBee 3.0 software architecture and the supported Zigbee device types. It also describes Zigbee Lighting and Occupancy device types and how to implement these devices on the FRDM-MCXW71, FRDM-MCXW72, MCX-W71-EVK, MCX-W72-EVK, FRDM-RW612 board, and RD-RW612-BGA hardware platforms provided by NXP. These platforms belong to the NXP provided MCXW71, MCXW72 and RW612 family of wireless microcontrollers.

The device software described in this manual can be used on the NXP MCXW71, MCXW72 and RW612 family of wireless microcontrollers.

The document is organized as follows:

- *Chapter 1: Introduction* introduces ZigBee device types and provides general guidance on implementing device types in ZigBee application software.
- *Chapter 2: ZigBee Base Device* describes the ZigBee Base Device (ZBD), including the associated functions and other resources.
- *Chapter 3: Lighting and Occupancy Device Types* describes the ZigBee Lighting and Occupancy (ZLO) device types, including the device software structures and functions.

Conventions

- **Bold** type font represents Files, folders, GUI elements, function names, and parameter types.
- *italics* type font represents function parameters.
- Courier New typeface represents Code fragments.

  Note: This is a sample **Note**. It highlights important additional information.

Parent topic:*Overview*

Acronyms and abbreviations

Table 1 lists the acronyms and abbreviations used in this document.

Acronym	Description
ACE	Ancillary Control Equipment
APDU	Application Protocol Data Unit
API	Application Programming Interface
BDB	Base Device Behavior
CIE	Control and Indicating Equipment
DRLC	Demand-Response and Load Control
HA	Home Automation
IAS	Intruder Alarm System
SDK	Software Development Kit
SE	Smart Energy
WD	Warning Device
ZBD	ZigBee Base Device
ZCL	ZigBee Cluster Library
ZLO	ZigBee Lighting and Occupancy
ZPS	ZigBee PRO Stack

Parent topic:*Overview*

Related documents and support resources

- ZigBee 3.0 Stack User Guide (JNUG3130)
- ZigBee Cluster Library (for ZigBee 3.0) User Guide (JNUG3132)
- Connectivity Framework Reference Manual
- 13-0402 Base Device Behavior Specification [from ZigBee Alliance]
- 15-0014 Lighting & Occupancy Device Specification [from ZigBee Alliance]
- 075123 rev 6 ZigBee Cluster Library Specification [from ZigBee Alliance]

To access online support resources such as SDKs, Application Notes, and User Guides, visit the Wireless Connectivity area of the NXP website:

https://www.nxp.com/products/wireless:WIRELESS-CONNECTIVITY

All NXP resources referred to in this manual can be found at the above address, unless otherwise stated.

Parent topic:*Overview*

Introduction The nodes of a ZigBee wireless network are based on device types defined by the ZigBee Alliance. Such a device type is a software entity that determines the functionality supported by a node. This chapter introduces ZigBee device types and describes related concepts that are required in programming software applications for ZigBee nodes.

Note: ZigBee device types have previously been collected together in market-specific application profiles, such as Home Automation. ZigBee 3.0 allows devices from different market sectors to exist in the same network. Therefore, application profiles are not so prevalent in ZigBee 3.0 but are still supported for backward compatibility.

ZigBee device types A device type is a software entity which defines the functionality of a ZigBee node. The device type defines a collection of clusters that make up this functionality. A cluster is therefore a basic building-block of device functionality. Some clusters are mandatory and some are optional. For example, the Thermostat device uses the Basic and Temperature Measurement clusters, and can also use one or more optional clusters.

Note: The clusters used by a device type are supplied in the ZigBee Cluster Library (ZCL). The ZCL is detailed in the *ZigBee Cluster Library (for ZigBee 3.0) User Guide (JNUG3132)*.

A device is an instance of a device type.

A network node can support more than one device type. The application for a device type runs on a software entity called an endpoint and each node can have up to 240 endpoints, numbered from 1.

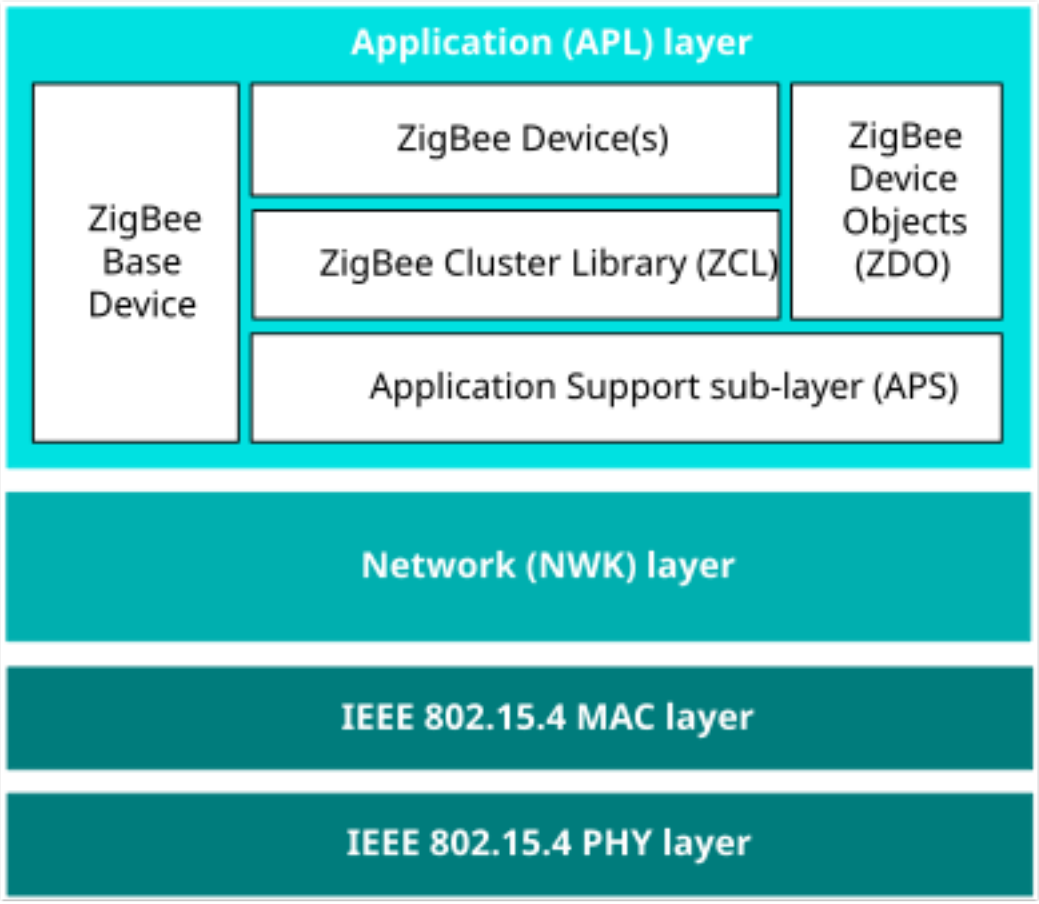
In addition, every ZigBee 3.0 node must employ the following devices:

- **ZigBee Base Device (ZBD):** This is a standard device type which handles fundamental operations such as commissioning. This device does not need an endpoint. The ZigBee Base Device is fully detailed in *Zigbee Base Device*.
- **ZigBee Device Objects (ZDO):** This represents the ZigBee node type (Coordinator, Router, or End Device) and has a number of communication roles. This device occupies endpoint 0.

The relative locations of the different devices are indicated in Software Architecture.

Parent topic:*Introduction*

Software architecture The figure below shows the basic ZigBee 3.0 software architecture, which illustrates the locations of the ZigBee devices.



For more detailed software architecture information, refer to the *ZigBee 3.0 User Guide (JNUG3130)*.

Parent topic: *Introduction*

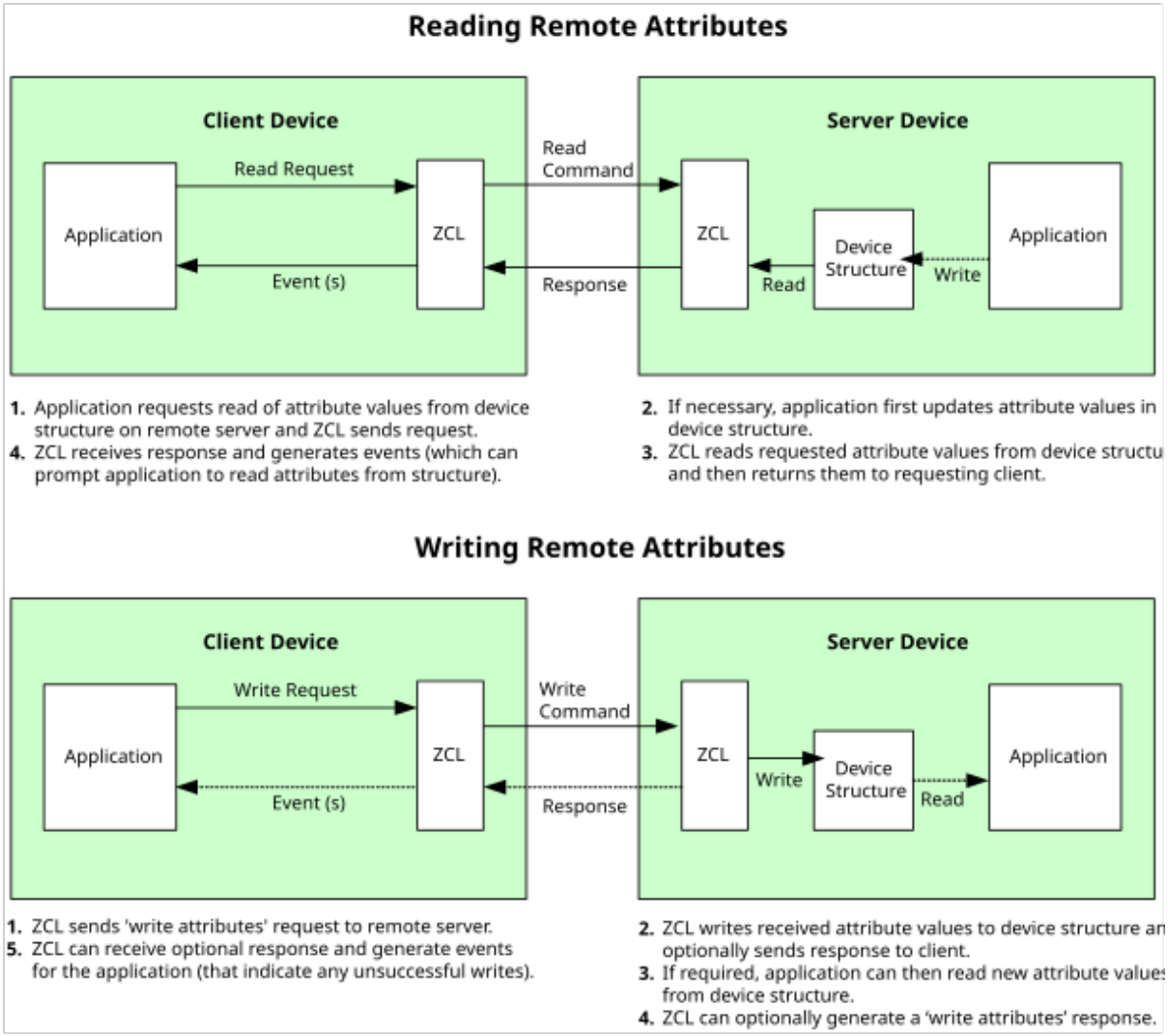
Shared device structure The basic operations in a ZigBee 3.0 network are concerned with reading and setting the attribute values of the clusters of a device. In each device, attribute values are exchanged between the application and the ZigBee Cluster Library (ZCL) by means of a shared structure. This structure is protected by a mutex (described in the *ZCL User Guide (JNUG3132)*). The structure for a particular device contains structures for the clusters supported by that device.

Note: In order to use a cluster which is supported by a device, the relevant option for the cluster must be specified at build-time - see Compile-time options.

A shared device structure may be used in either of the following ways:

- The local application writes attribute values to the structure, allowing the ZCL to respond to commands relating to these attributes.
- The ZCL parses incoming commands that write attribute values to the structure. The written values can then be read by the local application.

Remote read and write operations involving a shared device structure are illustrated in the figure below.



As shown in the above figure, the shared device structure is located on the server device, which hosts the cluster server to be accessed. The client device, which performs the remote access, hosts the corresponding cluster client. See

Note: If there are no remote attribute writes, the attributes of a cluster server (in the shared structure) on a device are maintained by the local application.

For more detailed descriptions of these operations, refer to the *ZCL User Guide (JNUG3132).*

Parent topic: *Introduction*

Endpoint callback functions A user-defined callback function must be provided for each endpoint used. The callback function is invoked when an event occurs (such as an incoming message) relating to the endpoint. The callback function is registered when the endpoint is registered using the registration function for the device type that the endpoint supports (see Device initialization) - for example, using the function **eZLO_RegisterOnOffLightEndPoint()** for an On/Off Light device (see On/Off light).

The endpoint callback function has the type definition given below:

```
typedef void (* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Here, *pCallBackEvent* is a pointer to the *pCallBackEvent* event.

Note: Events that do not have an associated endpoint are delivered via the general stack-supplied callback function **APP_vGenCallback()**. For example, stack leave and join events can be received by the application through this callback function. Stack events are described in the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

Parent topic: *Introduction*

Device initialization A ZigBee 3.0 application is initialized as described in the section "Forming and Joining a Network" of the *ZigBee 3.0 Stack User Guide (JNUG3130)*. In addition, some device initialization must be performed.

ZigBee devices must be initialized in the following order and steps:

1. In the header file **zcl_options.h**, enable the required compile-time options. These options include the clusters to be used by the device, the client/server status of each cluster and the optional attributes for each cluster. For more information on compile-time options, refer to Compile-time options.
2. In the application, create an instance of the device structure by declaring a file scope variable - for example:

```
tsZLO_DimmableLightDevicesDevice;
```

3. In the initialization part of the application, set up the device handled by your code, as follows:

a) Set the initial values of the cluster attributes to be used by the device - for example:

```
sDevice.sBasicCluster.u8StackVersion=1;
sDevice.sBasicCluster....
```

b) After calling **eZCL_Initialise()** and before calling **ZPS_eAplAfInit()**, register the device by calling the relevant device registration function - for example, **eZLO_RegisterDimmableLightEndPoint()**. In this function call, the device allocates a unique endpoint (in the range 1-240). In addition, its device structure is specified as well as a user-defined callback function is invoked when an event occurs relating to the endpoint (see Section 1.5). As soon as this function has been called, the shared device structure is read by another device.

c) After calling **ZPS_eAplAfInit()**, initialize and start the ZigBee Base Device (ZBD) by calling **BDB_vInit()** and then **BDB_vStart()**. Refer to Section 2.1 for more details of ZigBee Base Device initialization.

Note:

1. The set of endpoint registration functions for the different device types is detailed in the device type descriptions - for example, in *Chapter 3* for Lighting and Occupancy devices.
2. The device registration functions create instances of all the clusters used by the device. Thus, there is no need to call the individual cluster creation functions explicitly, for example, **eCLD_IdentifyCreateIdentify()** for the Identify cluster.

Parent topic:*Introduction*

Compile-time options Before a ZigBee 3.0 application is built, configure compile-time options in the header file **zcl_options.h** for the application.

Note:

1. Cluster-specific compile-time options are detailed in the cluster descriptions in the *ZCL User Guide (JNUG3132)*.

2. In addition, set compile-time options for the ZigBee Base Device in the file **bdb_options.h** - see Section 2.10.

Number of Endpoints An application must specify the highest numbered endpoint used by it - for example:

```
#define BDB_FB_NUMBER_OF_ENDPOINTS3
```

Normally, the endpoints starting at endpoint 1 are for application use, so the above case uses endpoints 1 to 3. It is possible, however, to use the lower numbered endpoints for non-application purposes, for example, to run other protocols on endpoints 1 and 2, and the application on endpoint 3. With BDB_FB_NUMBER_OF_ENDPOINTS set to 3, some storage is statically allocated for endpoints 1 and 2 but never used. Note that this define applies only to local endpoints - the application can refer to remote endpoints with numbers beyond the locally defined value of BDB_FB_NUMBER_OF_ENDPOINTS.

Manufacturer Code The ZCL allows a manufacturer code to define devices developed by a certain manufacturer. The value allocated to a manufacturer by the ZigBee Alliance is a 16-bit and is set as follows:

```
#define ZCL_MANUFACTURER_CODE 0x1037
```

The above example sets the manufacturer code to the default value of 0x1037 (which belongs to NXP) but manufacturers should set their own allocated value.

Enabled Clusters Enable all the required clusters in the options header file. For example, an application for an On/Off Light device that uses all the possible clusters requires the following definitions:

```
#define CLD_BASIC
#define CLD_IDENTIFY
#define CLD_GROUPS
#define CLD_SCENES
#define CLD_ONOFF
```

Server and Client options Many clusters have options that indicate whether the cluster acts as a server or a client on the local device. If the cluster is enabled using one of the above definitions, define the server/client status of the cluster. For example, to employ the Groups cluster as a server, include the following in the header file:

```
#define GROUPS_SERVER
```

Support for attribute Read/Write Compile read/write access to cluster attributes into the application explicitly. Separately enable the server and client sides of a cluster using the following macros in the options header file:

```
#define ZCL_ATTRIBUTE_READ_SERVER_SUPPORTED
#define ZCL_ATTRIBUTE_READ_CLIENT_SUPPORTED
#define ZCL_ATTRIBUTE_WRITE_SERVER_SUPPORTED
#define ZCL_ATTRIBUTE_WRITE_CLIENT_SUPPORTED
```

Each of the above definitions applies to all clusters used in the application.

Optional attributes Many clusters have optional attributes that may enable at compile time via the options header file - for example, the Basic cluster 'application version' attribute is enabled as follows:

```
#define CLD_BAS_ATTR_APPLICATION_VERSION
```

Parent topic:*Introduction*

ZigBee Base Device The **ZigBee Base Device** (ZBD) is a mandatory device on all nodes of a ZigBee 3.0 network. It exists alongside one or more other ZigBee device types on a node, but does not require an endpoint. The ZigBee Base Device provides a framework for the use of ZigBee device types. It implements basic functionality that all nodes require and ensures consistent behavior across all nodes, particularly regarding network creation, joining, and security.

The network commissioning and security functionality of the ZigBee Base Device is described in this chapter. The chapter also describes the NXP resources required to implement these features in ZigBee 3.0 applications on the **NXP hardware platforms**: **FRDM-MCXW71, FRDM-MCXW72, MCX-W71-EVK, and MCX-W72-EVK, FRDM-RW612, and RD-RW612-BGA**. These boards (hardware platforms) belong to NXP provided **MCXW71, MCXW72 and RW612 family of wireless microcontrollers**.

Detailed information about the ZigBee Base Device is provided in the *ZigBee Base Device Behavior Specification (13-0402)*, available from the ZigBee Alliance.

Initializing and starting the ZigBee Base Device Initialize the ZigBee Base Device in the application code using the function **BDB_vInit()**. Call this function after initializing the ZigBee PRO stack and after restoring the ZigBee Base Device attribute *bbdbNodeIsOnANetwork* from persistent storage.

Note:

1. **BDB_vInit()** internally calls the function **BDB_vSetKeys()**, which loads into memory the pre-configured link key from the file **bdb_link_keys.c**. Network security and the pre-configured link keys are described in Section 2.3.
2. The ZigBee Base Device requires a number of internal software times, the number defined by the macro BDB_ZTIMER_STORAGE. Therefore, when the application calls **ZTIMER_eInit()** to initialize the required software timers and allocate storage (array elements) for them, it must add BDB_ZTIMER_STORAGE timers for use by the ZigBee Base Device. This function must be called before **BDB_vInit()**. Software timers and their associated functions are described in the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

The ZigBee Base Device is started by calling the function **BDB_vStart()**. This function may or may not perform an action. It depends on the node type and whether the node was previously a member of a network. In both cases, the function finally invokes the callback function **APP_vBdbCallback()** with a suitable event.

If the node was not on a network: For a Router node that supports Touchlink commissioning (see Touchlink), the function selects a radio channel for the node from the set of primary channels for Touchlink specified in the BDBC_TL_PRIMARY_CHANNEL_SET bitmap (see Touchlink constants). Either the first channel of the specified set selects or, if the macro RAND_CHANNEL is set to TRUE (in the file **bdb_options.h**), a channel selects from the set at random.

For the Coordinator and other Router and End Device nodes, no action is taken. The application must then form a network (Coordinator or Router). Alternatively, it can join the node to

a network using one of the commissioning methods described in Network commissioning (End Device or Router).

In the above cases, the function generates a BDB_EVENT_INIT_SUCCESS event.

If the node was on a network: For Coordinator and Router nodes, no action is taken and the function generates a BDB_EVENT_INIT_SUCCESSevent.

For an End Device node, the function attempts to rejoin the node to the network. It performs a series of rejoin cycles, where each cycle comprises the following three rejoin attempts:

1. First attempt with the previously used network parameters (without network discovery)
2. Second attempt with network discovery on the set of primary channels specified in the *u32bdbPrimaryChannelSet* bitmap (attribute)
3. Third attempt with network discovery on the set of secondary channels specified in the *u32bdbSecondaryChannelSet* bitmap (attribute)

The channel bitmaps are ZigBee Base Device attributes, described in Section 2.5.1.

The above rejoin cycle performs up to a maximum of BDBC_IMP_MAX_REJOIN_CYCLES times, which is an implementation-specific ZigBee Base Device constant (see Section 2.5.2).

If a rejoin attempt is successful, the function generates the event BDB_EVENT_REJOIN_SUCCESS.

If all the rejoin attempts are unsuccessful, the function generates the event BDB_EVENT_REJOIN_FAILURE unless unsecured joins are enabled through the APS attribute *apsUseInsecureJoin*, in which case the function attempts a join through Network Steering (described in Network steering). The nature of the join depends on the value of the Extended PAN ID (EPID) set in the APS attribute *ApsUseExtendedPanid*:

- For a non-zero EPID, the node attempts to join the network with this EPID.
- For a zero EPID, the function attempts to join any available network.

This join is attempted with an automatic call to the function**BDB_eNsStartNwkSteering()**.

Parent topic:*ZigBee Base Device*

Network commissioning Network commissioning covers the following activities:

- Creating a network
- Allowing devices to join the network (through the local node)
- Joining a network
- Binding a local endpoint to an endpoint on a remote node
- Adding a remote node to a group

The commissioning activities performed by an individual node depend on the ZigBee node type (Coordinator, Router, End Device) and the commissioning modes that are enabled for the node. A number of different commissioning modes are available through the ZigBee Base Device. These modes are listed in the table below along with the commissioning activities that they support.
Functionality of Commissioning Modes

Commissioning mode	Functionality
Touchlink	• Creating a new network• Allowing other devices to join an existing network• Joining local device to an existing network
Network steering	• Allowing other devices to join an existing network• Joining local device to an existing network
Network formation	• Creating a new network
Finding and binding	• Binding a local endpoint to an endpoint on a remote node• Adding a remote node to a group

The commissioning modes are individually enabled/disabled via the attribute *u8bdbCommissioningMode*, as indicated in the table below. This attribute is a bitmap with a bit for each of four commissioning mode - a bit is to '1' to enable or '0' to disable the corresponding commissioning mode. Enumerations are available to enable the individual modes (set their bits to '1').

Commissioning Modes (configured via bdbCommissioningMode)

Bit	Commissioning mode	Enumeration
0	Touchlink	BDB_COMMISSIONING_MODE_TOUCHLINK
1	Network steering	BDB_COMMISSIONING_MODE_NWK_STEERING
2	Network formation	BDB_COMMISSIONING_MODE_NWK_FORMATION
3	Finding and binding	BDB_COMMISSIONING_MODE_FINDING_N_BINDING

The current commissioning state on a node is reflected in the attribute *ebdbCommissioningStatus*.

In the NXP implementation of the ZigBee Base Device, the individual commissioning modes are initiated under application control using supplied API functions. A commissioning mode is invoked by the application if the mode is enabled and the node type is relevant to the mode (for example, an End Device cannot perform Network Formation).

The commissioning modes are outlined in the subsections below. For detailed information on these modes, refer to the *ZigBee Base Device Behavior Specification (13-0402-08)*.

Note: A node is normally be prompted to enter commissioning by a user action, such as pressing a button on the node. This action may be on behalf of the node as a whole or a single endpoint on the node.

Touchlink Touchlink commissioning is used to form a new network and/or join a node to an existing network. Touchlink is initiated on a node called the 'initiator' which either is a member of an existing network or (if not) creates a new network. In both cases, the initiator joins a second node to the network, called the 'target' node.

Touchlink is provided as a cluster in the **ZigBee Cluster Library (ZCL)**. The initiator must support the Touchlink cluster as a client and the target node must support the cluster as a server. If it is required on a node, Touchlink commissioning must be enabled via the ZigBee Base Device attribute *u8bdbCommissioningMode*. For detailed information on the Touchlink Commissioning cluster and how to implement Touchlink, refer to the *(ZigBee Cluster Library User Guide (JNUG3132)[https://kex-daily.nxp.com/mcuxsdk-internal/latest/html/_static/wireless/Zigbee/JN-UG-3132-ZigBee3-Cluster-Library.pdf]*.

A 'Touchlink Pre-configured Link Key' is provided, which is used during the commissioning of a node into a secured network (see Network security).

If Touchlink commissioning is not successful, this is indicated by a status of NO_SCAN_RESPONSE through the attribute *ebdbCommissioningStatus* (all other states indicate success).

Parent topic: Network commissioning

Network steering Network Steering is used to join the local node to an existing network or allow other nodes to join a network via the local node.

If Network Steering is required on a node, enable it via the attribute *u8bdbCommissioningMode*. You can start Network Steering from your application by calling the function **BDB_eNsStartNwkSteering()**.

The path taken depends on whether the local node is already a member of a network, as indicated by the Boolean attribute *bbdbNodeIsOnANetwork*. In all cases, the outcome of Network Steering is indicated by events passed into the callback function **APP_vBdbCallback()**.

Node is already on a network When the node is a member of a network, it opens the network for other nodes to join for a fixed time period. It performs this by broadcasting a Management Permit Joining request (any node type can open the network in this way). This period is 180 seconds by default, but can be configured (in seconds) through the ZigBee Base Device constant **BDBC_MIN_COMMISSIONING_TIME** (see Constants). After initiating the above broadcast, the event BDB_EVENT_NWK_STEERING_SUCCESS is generated.

Node is not on a network When the node is not a member of a network and is a Router or End Device, it searches for a suitable network to join. If it finds one, it attempts to join the network, as follows:

1. The node performs a network discovery by scanning the primary set of radio channels specified through the *u32bdbPrimaryChannelSet* bitmap (attribute). If no open network is found, the network discovery is repeated on the secondary set of radio channels specified through the *u32bdbSecondaryChannelSet* bitmap (attribute). If still no network is found, the event BDB_EVENT_NO_NETWORK is generated and the Network Steering is abandoned.
2. If at least one open network is found, the node then attempts to join each discovered open network one by one, up to a maximum of **BDBC_MAX_SAME_NETWORK_RETRY_ATTEMPTS** times. If a network is successfully joined, the attribute *bbdbNodeIsOnANetwork* is set to TRUE. If there is no successful join following a scan of the primary channels, the scan is repeated (Step 1) on the secondary channels. If there is still no successful join following this scan, the BDB_EVENT_NWK_JOIN_FAILURE event is generated and the Network Steering is abandoned.
3. The joining node is authenticated and receives the network key from its parent. If the network being joined has centralized security and therefore a Trust Centre, the node unicasts a Node Descriptor request to the Trust Centre. The Node Descriptor received back is checked to ensure that the Trust Centre supports the ZigBee PRO stack version r21 or above. If so, the node performs the procedure for retrieving a new Trust Centre link key to replace its pre-configured link key. Failure at any point is indicated to the application by a BDB_EVENT_NWK_JOIN_FAILURE event.
4. On successful completion of the above steps, the joining node requests that the 'permit joining' time (for new nodes to join the network) is extended by **BDBC_MIN_COMMISSIONING_TIME** (180 s by default) and generates a BDB_EVENT_NWK_STEERING_SUCCESS event for the application.

Depending on the outcome of the above Network Steering process:

- If the node successfully joins a network, you may wish to bind the node to another node or add the node to a group, wherein it is necessary to continue to the Finding and Binding stage, described in Finding and Binding.
- If the node fails to join a network, you may wish to make sure that the desired network is open for joining and reinitiate this Network Steering procedure. If there is a Router node,

the application may opt to form its own distributed network, wherein it is necessary to continue to the Network Formation stage described in Network Formation.

Parent topic: Network commissioning

Network Formation Network Formation allows a new network to be created by a Coordinator or Router.

- A Coordinator forms a centralized security network (see Section 2.3.1) and activate its Trust Centre functionality.
- A Router forms a distributed security network (see Section 2.3.2).

If Network Formation is required on a node, enable it via the attribute *u8bdbCommissioningMode*. You can start Network Formation from your application by calling the function **BDB_eNfStartNwkFormation()**.

The node performs a scan of the primary set of radio channels specified through the *u32bdbPrimaryChannelSet* bitmap (attribute) to form a centralized or distributed network with a unique PAN ID on one of the free primary channels. If this network formation fails or the primary channel bitmap is set to zero, the node performs a scan of the secondary set of radio channels. these radio channels are specified through the *u32bdbSecondaryChannelSet* bitmap (attribute) to form a centralized or distributed network with a unique PAN ID on one of the free secondary channels.

During the formation of a distributed security network by a Router:

- The above channel scans start with the first channel of the relevant set and cover all the specified channels.
- If the macro RAND_CHANNEL is TRUE (in the application), a channel is selected at random from the scanned channels.
- The macro RAND_DISTRIBUTED_NWK_KEY is set to TRUE to choose a network key at random (but may be set to FALSE during application development in order to use a specific network key).
- The PAN ID and Extended PAN ID are allocated at random (but must not clash with the other networks operating in the neighbourhood).
- The 16-bit network address of the local is allocated at random.

In all cases, successful Network Formation is indicated by the event BDB_EVENT_NWK_FORMATION_SUCCESS through the callback function **APP_vBdbCallback()**. The unsuccessful Network Formation is indicated by the event BDB_EVENT_NWK_FORMATION_FAILURE.

If Network Formation is successful, the new network consists of just one node. Further nodes are added to the network using Network Steering (see Network steering) or Touchlink (see Touchlink).

Parent topic: Network commissioning

Finding and Binding Finding and Binding mode allows a node in the network to pair with another network node - for example, a new lamp may pair with a controller device, to allow control of the lamp. The objective of this commissioning mode is to bind an endpoint on a new node to a compatible endpoint on a remote node in the network (depending on the supported clusters). Alternatively, the new node is added to a group of nodes that are collectively controlled.

If it is required on a node, enable Finding and Binding via the attribute *u8bdbCommissioningMode*.

In Finding and Binding, a node have one of two roles:

- **Initiator:** This node either creates a (local) binding with a remote endpoint or requests that the remote endpoint is added to a group.
- **Target:** This node identifies itself, and receives and responds to requests from the initiator.

The intended outcome is a pairing between the initiator and the target. Usually, the initiator is a controller device. The path followed by the Finding and Binding process depends on whether the local endpoint is an initiator or a target.

Initiator Node Finding and Binding is started on an initiator node by calling the function **BDB_eFbTriggerAsInitiator()**. This function is called as the result of a user action on the node, such as a button-press. The initiator then remains in Finding and Binding mode for a fixed time-interval (in seconds) defined by the constant BDBC_MIN_COMMISSIONING_TIME. If Finding and Binding does not succeed within this time, the event BDB_EVENT_FB_TIMEOUT is generated and passed into the callback function **APP_vBdbCallback()**.

Once Finding and Binding starts, the initiator node searches for target endpoints by broadcasting an Identify Query command periodically with a period (in seconds) defined through the macro BDB_FB_RESEND_IDENTIFY_QUERY_TIME.

Note: Before each broadcast attempt, the event BDB_EVENT_FB_NO_QUERY_RESPONSE is generated and passed into **APP_vBdbCallback()**. This event allows the application to exit the current Finding and Binding process (see below).

If the initiator receives an Identify Query response from a remote endpoint, the application must pass the ZCL event BDB_E_ZCL_EVENT_IDENTIFY_QUERY to the Base Device using the function **BDB_vZclEventHandler()**. This event allows the Base Device to gather information about the identifying device by sending a Simple Descriptor request to the relevant endpoint. If the requested Simple Descriptor is successfully received back, the callback function checks this descriptor for clusters that match those on the initiator. The application is notified of via a BDB_EVENT_FB_HANDLE_SIMPLE_DESC_RESP_OF_TARGET event passed into **APP_vBdbCallback()**.

If there is at least one matching cluster, the initiator does one of the following:

- If binding is required (indicated by the *u16bdbCommissioningGroupID* attribute being equal to 0xFFFF), the initiator adds the remote endpoint to the local Binding table (but should first request the IEEE/MAC address of the remote node).
- If grouping is required (indicated by the *u16bdbCommissioningGroupID* attribute being equal to a 16-bit group address), the initiator requests that the target endpoint adds the group address to its Group Address table.

The application is notified of a successful binding or grouping via the following events:

- For a binding:
  - BDB_EVENT_FB_BIND_CREATED_FOR_TARGET for success
  - BDB_EVENT_FB_ERR_BINDING_FAILED for failure
- For a grouping:
  - BDB_EVENT_FB_GROUP_ADDED_TO_TARGET for success
  - BDB_EVENT_FB_ERR_GROUPING_FAILED for failure

At this point, the application can remotely stop identification mode (and therefore Finding and Binding) on the target node by calling the Identify cluster function **eCLD_IdentifyCommandIdentifyRequestSend()** to request that the identification mode period is set to zero.

A Finding and Binding process is stopped on the initiator endpoint using the function **BDB_vFbExitAsInitiator()**. This function is typically called in the callback function **APP_vBdbCallback()** as the result of a user action, such as a button-press or button-release.

Parent topic: Finding and Binding

Target Node Finding and Binding is started on a target node by calling the function **BDB_eFbTriggerAsTarget()**. This function is called as the result of a user action on the node, such as a button-press.

The target node then uses the Identify cluster to put itself into identification mode for a fixed time period. This period (in seconds) is determined by u16IdentifyTime, an Identify cluster attribute which is automatically set to the value of the constant BDBC_MIN_COMMISSIONING_TIME. In identification mode, the cluster responds to any received Identify Query commands, as well as other Finding and Binding commands. The node may also visually or audibly indicate that it is in identification mode. On exiting identification mode at the end of the above period, the cluster is no longer able to process Identify Query commands but the node is still able to service other commands from the initiator related to the binding/grouping. The Identify cluster is fully described in the *ZigBee Cluster Library User Guide (JNUG3132)*.

A target node can be brought out of the Finding and Binding process in either of the following ways:

- The local application can call the function **BDB_vFbExitAsTarget()** as the result of a user action, such as a button-press or button-release.
- The remote application (on the initiator) can call the Identify cluster function **eCLD_IdentifyCommandIdentifyRequestSend()** to request that the identification mode period is set to zero. To indicate to the Base Device that the identification process has ended, the application must pass the ZCL event BDB_E_ZCL_EVENT_IDENTIFY to the Base Device using the **BDB_vZclEventHandler()** function. This allows the Base Device to exit the 'Finding and Binding' process on the target endpoint.

Parent topic: Finding and Binding

Parent topic: Network commissioning

Out-Of-Band Commissioning A node is commissioned to a ZigBee network via out-of-band means - that is, not using IEEE802.15.4 packets operating in the radio channel used by the target network. For example, the out-of-band commissioning is conducted from another ZigBee device using inter-PAN packets (operating in a different radio channel) or by a commissioning device that uses NFC (Near Field Communication).

Out-of-band commissioning creates a new network by starting the Coordinator or to join a Router or End Device to an existing network. To do this, commissioning data must be sent to the node via an out-of-band means. This data includes details of the network (see Section 2.7.5). The application must pass the received commissioning data to the ZigBee Base Device and start out-of-band commissioning using the function **BDB_u8OutOfBandCommissionStartDevice()**. The data is then stored locally.

As part of the out-of-band commissioning of a node to an existing centralized network, the Trust Centre of the joined network must validate the new node by checking that the node contains appropriate data values, such as the correct network key and Trust Centre address. If such a validation request is received by the node, the required data values are obtained by the application in either of two ways:

- The function **BDB_vOutOfBandCommissionGetData()** is used to read the relevant data values. In this case, the application should encrypt the obtained network key before sending the data to the Trust Centre. The install code for the node should be used in this encryption.
- The function **BDB_eOutOfBandCommissionGetDataEncrypted()** is used to read the relevant data values and encrypt the obtained network key - therefore, the network key is delivered encrypted. The install code for the node used in this encryption must be specified in the function call. The application then sends the obtained data to the Trust Centre.

Once the Trust Centre receives the requested data, it decrypts the obtained network key using the function **BDB_bOutOfBandCommissionGetKey()** and then checks if the correct key is used. This function requires the install code for the new node, which must be supplied to the Trust Centre via out-of-band means (for example, via a keypad).

Security keys and install codes are described in Section 2.3.

Parent topic: Network commissioning

Parent topic:*ZigBee Base Device*

Network security The ZigBee Base Device supports the following network security modes:

- Centralized security
- Distributed security

These security modes are described in the subsections below:

All Router and End Device nodes should support both centralized security and distributed security by adapting to the security scheme employed by the network that they join. A Co-ordinator supports only centralized security.

When the application calls **BDB_vInit()**, this function internally calls the function **BDB_vSetKeys()**. This function loads the appropriate pre-configured link key, depending on whether the node type supports centralized and/or distributed security. The pre-configured link keys are defined in the file **bdb_link_keys.c**.

Centralized security networks A centralized security network is formed by a Coordinator, which also acts as the Trust Centre for the network. When a node attempts to join the network, it is authenticated by this Trust Centre before it is allowed into the network.

For participation in centralized security networks, all nodes must be pre-configured with a link key. This key is used to encrypt the network key when passing it from the Trust Centre to a newly joined node. When a node joins a network with centralized security, the ZigBee Base Device automatically uses the relevant pre-configured link key. Similar is the case for a Coordinator that forms a new centralized security network.

The following key types can be pre-configured for centralized security:

- **Default Global Trust Centre Link Key:** This key is factory-programmed into all nodes and is used to encrypt communications between the Trust Centre and a joining node.
- **Touchlink Pre-configured Link Key:** This key is factory-programmed into all nodes that can employ Touchlink commissioning and is used to encrypt communications between the Router parent and a joining node. The Touchlink Pre-configured Link Key can be one of three types:
  - Development key, used during development before ZigBee certification
  - Master key, used after successful ZigBee certification
  - Certification key, used during ZigBee certification testing
    - The link key used in the final products should be a 'master key', which results from the successful ZigBee certification of the product.
- **Install Code-derived Pre-configured Link Key:** This key is derived by the ZigBee stack from a random install code which is assigned to each Router and End Device node in the factory. The install code is factory-programmed into the node but provided to the Trust Centre via out-of-band means when the node is commissioned. The use of install codes is described in more detail below.

Install Codes An install code is used to create an initial link key employed in commissioning an individual node into a centralized security network. An install code is assigned to the node in the factory. It is a random code but is not necessarily unique (the same install code may be randomly generated for more than one node). The ZigBee stack derives a link key from the install code using a Matyas-Meyer-Oseas hash function. The install code is factory-programmed into the node and also accompanies the node (for example: in printed form) when it leaves the factory. The process of using an install code to commission a node is outlined below.

In the factory:

1. An install code is randomly generated for the individual node.
2. The install code is programmed into the node.
3. A pre-configured link key is derived from the install code by the ZigBee stack.
4. The install code is shipped with the node (by some unspecified means).

During installation:

5. The install code that was shipped with the node is installed into the Co-ordinator/Trust Centre.
6. The pre-configured link key is derived from the install code by the ZigBee stack of the Co-ordinator/Trust Centre.
7. The Trust Centre and node then use the pre-configured link key in joining the node to the network (for example: to encrypt/decrypt the network key).

More detailed information about install codes are available in the *ZigBee Base Device Behavior Specification (13-0402-08)*.

Parent topic: Network security

Distributed security networks A distributed security network is formed by a Router and does not have a Trust Centre. It consists only of Routers and End Devices. When a node attempts to join the network, it is authenticated by its Router parent before it is allowed into the network.

For participation in distributed security networks, all Router and End Device nodes must be pre-configured with a link key. This key is used to encrypt the network key when passing it from a Router parent to a newly joined node. When a Router or End Device joins a network with distributed security, the ZigBee Base Device automatically uses the relevant pre-configured link key. Similar is the case for a Router that forms a new distributed security network.

The following key types can be pre-configured for distributed security:

- **Distributed Security Global Link Key:** This key is factory-programmed into all nodes and is used to encrypt communications between the Router parent and a joining node.
- **Touchlink Pre-configured Link Key:** This key is factory-programmed into all nodes that can employ Touchlink commissioning and is used to encrypt communications between the Router parent and a joining node. The Touchlink Pre-configured Link Key can be one of three types:
  - Development key, used during development before ZigBee certification
  - Master key, used after successful ZigBee certification
  - Certification key, used during ZigBee certification testing
    - The link key used in the final products should be a 'master key', which results from the successful ZigBee certification of the product.

Parent topic: Network security

Parent topic:*ZigBee Base Device*

ZigBee base device rejoin handling For a Router or End Device, there are instances in which the ZigBee PRO stack initiates a network rejoin attempt and include:

- A Router or End Device which receives a 'leave with rejoin' request.
- An End Device which polls its parent for data but fails to receive a response.

The ZigBee Base Device handles the stack events that result from this rejoin attempt:

- If the stack event ZPS_EVENT_NWK_FAILED_TO_JOIN is received to indicate an unsuccessful rejoin, the ZigBee Base Device makes a series of rejoin attempts as described for the case "If the node was in a network" in Section 2.1. If a rejoin attempt is successful, the event BDB_EVENT_REJOIN_SUCCESS is generated to notify the application. If all rejoins are unsuccessful, the event BDB_EVENT_REJOIN_FAILURE is generated unless unsecured joins are enabled, in which case a join through Network Steering is attempted.
- If the stack event ZPS_EVENT_NWK_JOINED_AS_ROUTER or ZPS_EVENT_NWK_JOINED_AS_END_DEVICE is received to indicate a successful rejoin, the event BDB_EVENT_REJOIN_SUCCESS is generated to notify the application.

Parent topic:*ZigBee Base Device*

Attributes and Constants

Attributes The attributes of the ZigBee Base Device are contained in the structure BDB_tsAttrib, shown below:

```
typedefstruct
{
    uint16 u16bdbCommissioningGroupID;
    uint8 u8bdbCommissioningMode;
    BDB_teCommissioningStatusebdbCommissioningStatus;
    uint64 u64bdbJoiningNodeEui64;
    uint8 au8bdbJoiningNodeNewTCLinkKey[16];
    bool_tbbdbJoinUsesInstallCodeKey;
    const uint8 u8bdbNodeCommissioningCapability;
    bool_tbbdbNodeIsOnANetwork;
    uint8 u8bdbNodeJoinLinkKeyType;
    uint32 u32bdbPrimaryChannelSet;
    uint8 u8bdbScanDuration;
    uint32 u32bdbSecondaryChannelSet;
    uint8 u8bdbTCLinkKeyExchangeAttempts;
    uint8 u8bdbTCLinkKeyExchangeAttemptsMax;
    uint8 u8bdbTCLinkKeyExchangeMethod;
    uint8 u8bdbTrustCenterNodeJoinTimeout;
    bool_tbbdbTrustCenterRequireKeyExchange;
    bool_tbTLStealNotAllowed;
    bool_tbLeaveRequested;
}BDB_tsAttrib;
```

For information on compile-time options, refer to Compile-time Options. The attributes are written to or read at run time through the above structure. The ZigBee Base Device attribute values are initialized at compile time in the **bdb_options.h** file using the macros listed in the table below.

Note: Both bTLStealNotAllowed and bLeaveRequested are NXP proprietary variables and not ZigBee attributes. **Table: ZBD Attributes and Initialization Macros**

Attribute	Initialization Macro
u16bdbCommissioningGroupID	BDB_COMMISSIONING_GROUP_ID
u8bdbCommissioningMode	BDB_COMMISSIONING_MODE
ebdbCommissioningStatus	BDB_COMMISSIONING_STATUS
u64bdbJoiningNodeEui64	BDB_JOINING_NODE_EUI64
au8bdbJoiningNodeNewTCLinkKey[16]	-
bbdbJoinUsesInstallCodeKey	BDB_JOIN_USES_INSTALL_CODE_KEY
u8bdbNodeCommissioningCapability	-
bbdbNodeIsOnANetwork	-
u8bdbNodeJoinLinkKeyType	BDB_NODE_JOIN_LINK_KEY_TYPE
u32bdbPrimaryChannelSet	BDB_PRIMARY_CHANNEL_SET
u8bdbScanDuration	BDB_SCAN_DURATION
u32bdbSecondaryChannelSet	BDB_SECONDARY_CHANNEL_SET
u8bdbTCLinkKeyExchangeAttempts	BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS
u8bdbTCLinkKeyExchangeAttemptsMax	BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS_MAX
u8bdbTCLinkKeyExchangeMethod	BDB_TC_LINK_KEY_EXCHANGE_METHOD
u8bdbTrustCenterNodeJoinTimeout	BDB_TRUST_CENTER_NODE_JOIN_TIMEOUT
bbdbTrustCenterRequireKeyExchange	BDB_TRUST_CENTER_REQUIRE_KEYEXCHANGE
bTLStealNotAllowed	-
bLeaveRequested	-

The attributes are individually described below. For further details, refer to the *ZigBee Base Device Behavior Specification (13-0402-08)*.

u16bdbCommissioningGroupID This attribute can only be used on a Finding and Binding initiator endpoint. It contains the identifier of the group in which the initiator puts the target endpoints. If it is equal to 0xFFFF, individual (rather than group) bindings are created. The value of this attribute can be initialized at compile time using the macro BDB_COMMISSIONING_GROUP_ID.

Use of this attribute requires Finding and Binding to be enabled in the u8bdbCommissioningMode attribute.

The Finding and Binding commissioning mode is described in Finding and Binding.

u8bdbCommissioningMode This attribute is a bitmap used to indicate which commissioning modes are enabled on an endpoint. Each bit corresponds to a commissioning mode and is set (to '1') when the mode is enabled - this means that the node is able to implement this commissioning mode, if necessary. The value of this attribute can be initialized at compile time using the macro BDB_COMMISSIONING_MODE. The bitmap, along with the enumerations used to set the bits is illustrated in the table below.

Table: u8bdbCommissioningMode bitmap

Bit	Commissioning Mode	Enumeration
0	Touchlink	BDB_COMMISSIONING_MODE_TOUCHLINK
1	Network Steering	BDB_COMMISSIONING_MODE_NWK_STEERING
2	Network Formation	BDB_COMMISSIONING_MODE_NWK_FORMATION
3	Finding and Binding	BDB_COMMISSIONING_MODE_FINDING_N_BINDING
4-7	Reserved (set to '0')	-

The commissioning modes are described in Network commissioning.

Note: The attribute is used on all node types. However, in order to enable a commissioning mode, it must be available on the node, as indicated through the attribute

u8bdbNodeCommissioningCapability. The enabled commissioning modes are a subset of the commissioning capabilities of the node.

ebdbCommissioningStatus This attribute indicates the status of the commissioning process that is underway on an endpoint. The attribute takes one of the values defined in the BDB_teCommissioningStatus enumerations (see BDB_teCommissioningStatus). The attribute is used on all node types. The value of this attribute is updated internally by the ZigBee Base Device implementation, but can be read by the application.

u64bdbJoiningNodeEui64 This attribute contains the 64-bit IEEE/MAC address of a node that is in the process of joining a centralized security network. It is used on the network Coordinator only. The value of this attribute is updated internally by the ZigBee Base Device implementation.

au8bdbJoiningNodeNewTCLinkKey This attribute contains a new link key for use with a node that is joining the network but has not yet been granted full network membership. The value of this attribute is updated internally by the ZigBee Base Device implementation (on a joining node and its parent).

bbdbJoinUsesInstallCodeKey This attribute indicates whether a pre-configured link key must be available for a node before it is allowed to join the network - this may be a pre-installed link key or may be derived from an install code. A value of TRUE means that a link key is required, while FALSE means that a link key is not required. It is used on the network Co-ordinator/Trust Centre only. The value of this attribute can be initialized at compile time using the macro BDB_JOIN_USES_INSTALL_CODE_KEY. By default, the attribute should be set to FALSE. The attribute is not used by the ZigBee Base Device. If the attribute is set to TRUE, it is the responsibility of the application to handle this functionality directly and to set the required key (see the attribute u8bdbNodeJoinLinkKeyType).

u8bdbNodeCommissioningCapability This attribute is a bitmap indicating the commissioning capabilities of the node. Each bit corresponds to a commissioning capability and is set (to '1') if the capability is present. The attribute is used on all node types. The application cannot write directly to these bits - they are set according to the options defined in the application makefile. The bitmap and the related makefile options are detailed in the Table below.

Table: bdbCommissioningCapability Bitmap

Bit	Capability	Makefile Options
0	Network Steering	Is set to '1' if BDB_SUPPORT_NWK_STEERING is defined
1	Network Formation	Is set to '1' if BDB_SUPPORT_NWK_FORMATION is defined
2	Finding and Binding	Is set to '1' if either of the following is defined:• BDB_SUPPORT_FIND_AND_BIND_INITIATOR• BDB_SUPPORT_FIND_AND_BIND_TARGE
3	Touchlink	Is set to '1' if any of the following is defined:• BDB_SUPPORT_TOUCHLINK_INITIATOR_END_DEVICE• BDB_SUPPORT_TOUCHLINK_IN
4-7	Reserved (set to '0')	-

The above commissioning modes are described in Network commissioning.

Note: In order to use one of the available commissioning modes, the mode must also be enabled through the attribute u8bdbCommissioningMode. The enabled commissioning modes are a subset of the commissioning capabilities of the node.

bbdbNodeIsOnANetwork This attribute indicates whether the local node is a member of a network. A value of TRUE means that it is in a network (but not necessarily bound to any remote nodes), while FALSE means that it is not in a network. The attribute is used on all node types but the ZigBee Base Device does not maintain it. The application is responsible for persisting the attribute value and initializing the attribute following a power-cycle (before any other ZigBee Base Device functions are called).

u8bdbNodeJoinLinkKeyType This attribute indicates the type of link key with which the node is able to decrypt the encrypted network key received over-air when the node joins a new network. The attribute is used by Router and End Device nodes. The attribute values and the corresponding link key types as well as the macros use to define the link keys are listed in the table below. **Table: bdbNodeJoinLinkKeyType Values and Macros**

Value	Link Key Type	Link Key Definition Macro
0x00	Default global trust Centre link key	DEFAULT_GLOBAL_TRUST_CENTER_LINK_KEY
0x01	Distributed security global link key	DISTRIBUTED_SECURITY_GLOBAL_LINK_KEY
0x02	Install code derived pre-configured link key	INSTALL_CODE_DERIVED_PRECONFIGURED_LINK _ KEY
0x03	Touchlink pre-configured link key	TOUCHLINK_PRECONFIGURED_LINK_KEY

u32bdbPrimaryChannelSet This attribute specifies the primary (first-choice) set of 2.4 GHz radio channels that are used in channel scans. The attribute is a bitmap in which each bit corresponds to a channel and should be set to '1' if the channel is to be included in a scan. The bit number corresponds directly to the channel number - for example, bit 11 corresponds to the 2.4 GHz channel 11 and bit 26 corresponds to channel 26. This attribute is used on all node types. The value of this attribute can be initialized at compile time using the macro BDB_PRIMARY_CHANNEL_SET.

u8bdbScanDuration This attribute determines the duration of a scan operation per 2.4 GHz radio channel. The actual scan duration is calculated from the attribute value as follows:

aBaseSuperframeDuration x (2bdbScanDuration + 1)

where aBaseSuperframeDuration is defined in the IEEE 802.15.4 specification.

The attribute is used on all node types. The value of this attribute is taken from the Scan Duration Time set in the ZPS Configuration Editor.

u32bdbSecondaryChannelSet This attribute specifies the secondary (second-choice) set of 2.4 GHz radio channels that are used in channel scans. This channel set is used if the scan of primary channels is unsuccessful. The attribute is a bitmap in which each bit corresponds to a channel. In order to include the channel in a scan, set each bit to '1'. The bit number corresponds directly to the channel number - for example, bit 11 corresponds to the 2.4 GHz channel 11 and bit 26 corresponds to channel 26. If a scan of secondary channels is not required, the attribute should be set to zero. The attribute is used on all node types. The value of this attribute can be initialized at compile time using the macro BDB_SECONDARY_CHANNEL_SET.

u8bdbTCLinkKeyExchangeAttempts This attribute indicates the number of attempts to request a new link key that were made when the node joined the network. The attribute is used on Router and End Device nodes. The value of this attribute can be initialized at compile time using the macro BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS.

u8bdbTCLinkKeyExchangeAttemptsMax This attribute specifies the maximum number of key establishment attempts that are made before key establishment is abandoned when the node joins a new network. The attribute is used on Router and End Device nodes. The value of this attribute can initialized at compile time using the macro BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS_MAX.

u8bdbTCLinkKeyExchangeMethod This attribute specifies the method that was used to obtain a new link key when the node joined the network. The attribute values and corresponding methods are listed in the table below. This attribute is used on Router and End Device nodes.
Table: bdbTCLinkKeyExchangeMethod Values

Value	Key Exchange Method
0x00	APS Request Key
0x01	Certificate Based Key Exchange (CBKE)

The value of this attribute can be initialized at compile time using the macro BDB_TC_LINK_KEY_EXCHANGE_METHOD. It should be initialized to 0x00 (APS Request Key).

u8bdbTrustCenterNodeJoinTimeout This attribute specifies a timeout (in seconds) for the Trust Centre to delete the Trust Centre-generated link key for a newly joined node when key establishment with the node was unsuccessful. The attribute is used on the network Coordinator/Trust Centre only. The value of this attribute can be initialized at compile time using the macro BDB_TRUST_CENTER_NODE_JOIN_TIMEOUT.

bbdbTrustCenterRequireKeyExchange This attribute specifies whether the Trust Centre requires a joining node to replace its initial link key with a new link key generated by the Trust Centre. A value of TRUE means that the joining node successfully completes the link key exchange procedure and failure to do so results in the node being removed from the network. A value of FALSE means that the joining node is allowed to remain in the network even if it does

not successfully complete the link key exchange procedure. The attribute is used on the network Co-ordinator/Trust Centre only. The value of this attribute is initialized at compile time using the macro BDB_TRUST_CENTER_REQUIRE_KEYEXCHANGE. It should be initialized according to the Trust Centre policy that is implemented in the network - by default, set it to FALSE for backward compatibility.

bTLStealNotAllowed This attribute is an NXP proprietary flag which the application can set to prevent Touchlink commissioning commands from another node in a different network from 'stealing' the local node. Clearing the flag allows the node to be stolen, in which case it leaves the current network and either joins the other network or forms a new distributed network, as instructed by Touchlink initiator.

bLeaveRequested This attribute is an NXP proprietary flag which the application should only read and not write to. If Touchlink commissioning operations cause the ZigBee Base Device to initiate a network leave, then this flag is set by the Base Device. When a ZPS_EVENT_NWK_LEAVE_CONFIRM stack event is generated, the application should read this flag. If it reads as TRUE, the application should not handle the event (since the ZigBee Base Device handles it).

Parent topic: Attributes and Constants

Constants The ZigBee Base Device constants are divided into two categories:

- Constants used on all nodes - see General constants.
- Constants used on nodes that support Touchlink - see Touchlink constants.

General constants The table below lists the ZigBee Base Device constants that can be used on all nodes and also shows the corresponding macros used to define the constant values in the **bdb_options.h** file.

Table: ZBD General Constants and Macros

Constant	Macro
bdbcMaxSameNetworkRetryAttempts	BDBC_MAX_SAME_NETWORK_RETRY_ATTEMPTS
bdbcMinCommissioningTime	BDBC_MIN_COMMISSIONING_TIME
bdbcRecSameNetworkRetryAttempts	BDBC_REC_SAME_NETWORK_RETRY_ATTEMPTS
bdbcTCLinkKeyExchangeTimeout	BDBC_TC_LINK_KEY_EXCHANGE_TIMEOUT

bdbcMaxSameNetworkRetryAttempts This constant specifies the maximum number of join or key exchange attempts that the node can make on the same network. The value of this constant is defined using the macro BDBC_MAX_SAME_NETWORK_RETRY_ATTEMPTS and should be set to 10 (as recommended in the *ZigBee BDB Specification*).

bdbcMinCommissioningTime This constant specifies the minimum time-interval (in seconds) for which a network is open to allow new nodes to join or for a device to identify itself. The value of this constant is defined using the macro BDBC_MIN_COMMISSIONING_TIME and should be set to 180 (as recommended in the *ZigBee BDB Specification*).

bdbcRecSameNetworkRetryAttempts This constant specifies the recommended number of join or key exchange attempts that the node can make on the same network. The value of this constant is defined using the macro BDBC_REC_SAME_NETWORK_RETRY_ATTEMPTS and should be set to 3 (as recommended in the *ZigBee BDB Specification*).

bdbcTCLinkKeyExchangeTimeout This constant specifies the maximum time (in seconds) for which a joining node will wait for a response after an APS key request has been sent to the Trust Centre. The value of this constant is defined using the macro BDBC_TC_LINK_KEY_EXCHANGE_TIMEOUT and should be set to 5 (as recommended in the *ZigBee BDB Specification*).

Parent topic: Constants

Touchlink constants The table below lists the ZigBee Base Device constants that can be used on nodes that support Touchlink commissioning and also shows the corresponding macros used to define the constant values in the **bdb_options.h** file.

ZBD Touchlink Constants and	Macros
Constant	**Macro**
bdbcTLInterPANTransIdLifetime	BDBC_TL_INTERPAN_TRANS_ID_LIFETIME
bdbcTLMinStartupDelayTime	BDBC_TL_MIN_STARTUP_DELAY_TIME
bdbcTLPrimaryChannelSet	BDBC_TL_PRIMARY_CHANNEL_SET
bdbcTLRxWindowDuration	BDBC_TL_RX_WINDOW_DURATION
bdbcTLScanTimeBaseDuration	BDBC_TL_SCAN_TIME_BASE_DURATION_MS
bdbcTLSecondaryChannelSet	BDBC_TL_SECONDARY_CHANNEL_SET

bdbcTLInterPANTransIdLifetime This constant specifies the maximum length of time (in seconds) that an inter-PAN transaction ID remains valid. The value of this constant is defined using the macro BDBC_TL_INTERPAN_TRANS_ID_LIFETIME and should be set to 8 (as recommended in the ZigBee BDB Specification).

bdbcTLMinStartupDelayTime This constant specifies the length of time (in seconds) that a Touchlink initiator waits for the target to complete its network start-up procedure. The value of this constant is defined using the macro BDBC_TL_MIN_STARTUP_DELAY_TIME and should be set to 2 (as recommended in the ZigBee BDB Specification).

bdbcTLPrimaryChannelSet This constant specifies the bitmap for the primary (first-choice) set of 2.4 GHz radio channels that is used for a non-extended Touchlink scan. The value of this constant is defined using the macro BDBC_TL_PRIMARY_CHANNEL_SET and should be set to 0x02108800, corresponding to channels 11, 15, 20 and 25 (as recommended in the ZigBee BDB Specification).

bdbcTLRxWindowDuration This constant specifies the maximum duration (in seconds) for which the radio receiver of node remains enabled during Touchlink commissioning, in order to receive responses. The value of this constant is defined using the macro BDBC_TL_RX_WINDOW_DURATION and should be set to 5 (as recommended in the ZigBee BDB Specification).

bdbcTLScanTimeBaseDuration This constant specifies the base duration (in milliseconds) for which the radio receiver of node remains enabled after transmitting a scan request during a Touchlink scan operation, in order to receive responses. The value of this constant is defined using the macro BDBC_TL_SCAN_TIME_BASE_DURATION_MS and should be set to 250 (as recommended in the ZigBee BDB Specification).

bdbcTLSecondaryChannelSet This constant specifies the bitmap for the secondary (second-choice) set of 2.4 GHz radio channels that is used for an extended Touchlink scan. It should contain the channels that remain from those specified in *bdbcTLPrimaryChannelSet*. The value of this constant is defined using the macro BDBC_TL_SECONDARY_CHANNEL_SET and should be set to 0x07FFF800 XOR BDBC_TL_PRIMARY_CHANNEL_SET.

Parent topic: Constants

Parent topic: Attributes and Constants

Parent topic:*ZigBee Base Device*

Functions This section details the C functions that are provided for the ZigBee Base Device. The functions are listed below along with the references to their descriptions.

1. BDB_vInit
2. BDB_vSetKeys
3. BDB_vStart
4. BDB_eNfStartNwkFormation
5. BDB_eNsStartNwkSteering
6. BDB_eFbTriggerAsInitiator
7. BDB_vFbExitAsInitiator
8. BDB_eFbTriggerAsTarget
9. BDB_vFbExitAsTarget
10. BDB_bIsBaseIdle
11. BDB_u8OutOfBandCommissionStartDevice
12. BDB_vOutOfBandCommissionGetData
13. BDB_eOutOfBandCommissionGetDataEncrypted
14. BDB_bOutOfBandCommissionGetKey

Note:

1. The application must provide a user-defined callback function, **APP_vBdbCallback()**, to handle ZigBee Base Device events. The prototype for this function is given in Events.
2. The ZigBee Base Device supplies the callback function **BDB_vZclEventHandler()**, which handles certain ZCL events during the Finding and Binding process, as indicated in Finding and Binding.

BDB_vInit

```
void BDB_vInit(BDB_tsInitArgs *psInitArgs);
```

Description This function initializes the ZigBee Base Device (ZBD) and must be the first ZigBee Base Device function called in your code. The function must be called after initializing the ZigBee PRO stack via a call to **ZPS_eAplAfInit()**. The ZigBee Base Device attribute *bbdbNodeIsOnANetwork* is restored from persistent storage (if relevant) before calling this function.

Note: Before calling this function, the application must initialize the required ZigBee software timers using the function **ZTIMER_eInit()** from the ZigBee PRO Stack libraries. In doing so, it must add a number of timers for internal use by the ZigBee Base Device, where this number is defined by the macro BDB_ZTIMER_STORAGE.

The initialization performed by this function includes the following:

- Sets the ZigBee Base Device attributes to their default values, unless other values are defined by the application in the file **bdb_options.h**.
- Registers the ZigBee Base Device message queue passed into this function - this message queue is used by the ZigBee Base Device to capture stack events.
- Calls **BDB_vSetKeys()** to set the initial pre-configured security keys (defined in the file **bdb_link_keys.c**), according to the node type:
  - For a Coordinator, the Default Global Trust Centre Link Key is set.
  - For a Router or End Device, both the Default Global Trust Centre Link Key and Distributed Security Global Link Key are set.
- Opens timers for ZigBee Base Device internal use

For more information on the security keys, refer to Section 2.3.

Parameters

- *psInitArgs*: Handle of the ZigBee Base Device event queue

Returns

- None

Parent topic: Functions

BDB_vSetKeys

```
void BDB_vSetKeys(void);
```

Description This function loads into memory the appropriate pre-configured link key on the local node for the initial security state of the node. The function is automatically called by **BDB_vInit()**. However, it must be called explicitly to restore the link keys after a reset which removes the keys from memory.

The type of link key that is loaded depends on the node type, as follows:

- On a Coordinator, the Default Global Trust Centre Link Key is loaded for participation in a centralized security network.
- On a Router or End Device, both of the following keys are loaded:
  - Default Global Trust Centre Link Key for participation in a centralized security network.
  - Distributed Security Global Link Key for participation in a distributed security network.

The pre-configured link keys are defined in the file **bdb_link_keys.c**, from where they are loaded.

Network security is described in Section 2.3.

Parameters

- None

Returns

- None

Parent topic: Functions

BDB_vStart

```
void BDB_vStart(void);
```

Description This function starts the ZigBee Base Device (ZBD) and must be called after **BDB_vInit()** and just before the application enters the main loop (for example: **APP_vMainLoop()**).

Depending on the node type and whether the node was previously a member of a network, the function may or may not perform an action. In either case, the function invokes the callback function **APP_vBdbCallback()** with a suitable event.

- If the node was not in a network:
  - For a Router node that supports Touchlink commissioning, the function selects a radio channel for the node from the set of primary channels defined for Touchlink.
  - For other Router, Coordinator, and End Device nodes, no action is taken and the application then must join the node to a network.
    - In the above cases, the function generates the event BDB_EVENT_INIT_SUCCESS.
- If the node was in a network:
  - For Coordinator and Router nodes, no action is taken by the function and the event BDB_EVENT_INIT_SUCCESS is generated.
  - For an End Device node, a series of rejoin attempts are performed. If a rejoin attempt is successful, the event BDB_EVENT_REJOIN_SUCCESS is generated. If all rejoins are unsuccessful, the event BDB_EVENT_REJOIN_FAILURE is generated unless unsecured joins are enabled, in which case a join through Network Steering is attempted.

The above actions are described in more detail in Section 2.1.

Parameters

- None

Returns

- None

Parent topic: Functions

BDB_eNfStartNwkFormation

```
BDB_teStatus BDB_eNfStartNwkFormation(void);
```

Description This function starts the Network Formation process and, if necessary, must be called after **BDB_vStart()**. If it is potentially required on a node, Network Formation must be enabled via the attribute u8bdbCommissioningMode.

The function can be called only on a Coordinator or Router:

- If called on a Coordinator, a centralized security network is formed.
- If called on a Router, a distributed security network is formed.

The above network types are described in Network security.

Once Network Formation starts, the function returns and the eventual outcome of the Network Formation process is indicated by an asynchronous event - one of the following:

- BDB_EVENT_NWK_FORMATION_SUCCESS if a centralized or distributed network has been successfully formed.
- BDB_EVENT_NWK_FORMATION_FAILURE if a network has not been successfully formed.

Network Formation is described in more detail in Network Formation.

Parameters

- None

Returns

- BDB_E_SUCCESS

  (Network Formation has been successfully started)*.
- BDB_E_ERROR_INVALID_PARAMETER

  (End Device has attempted Network Formation).
- BDB_E_ERROR_NODE_IS_ON_A_NWK

  (node is already in a network).

The eventual outcome is indicated by a BDB_EVENT_NWK_FORMATION_SUCCESS or BDB_EVENT_NWK_FORMATION_FAILURE event, as described above.

Parent topic: Functions

BDB_eNsStartNwkSteering

```
BDB_teStatus BDB_eNsStartNwkSteering(void);
```

Description This function starts the Network Steering process and, if necessary, must be called after **BDB_vStart()**. If it is potentially required on a node, enable Network Steering via the attribute u8bdbCommissioningMode.

The actions performed by this function depend on whether the local node is already a member of a network:

- When the node is already in a network and is a Coordinator or Router, it opens up the network for other nodes to join. This action is for a fixed time-interval of 180 seconds by default, but this interval can be configured (in seconds) using the macro BDBC_MIN_COMMISSIONING_TIME in the **bdb_options.h** file.
- When the node is not already in a network, it searches for a suitable network to join. If it finds one, attempts to join the network. Once a node has joined the network, the node is authenticated and receives the network key from its parent. If the network has a Trust Centre, the node may then replace its pre-configured link key with one generated and supplied by the Trust Centre.

Once Network Steering starts, the function returns and the eventual outcome of the Network Steering process is indicated by an asynchronous event - one of the following:

- BDB_EVENT_NWK_STEERING_SUCCESS if Network Steering has been completed successfully.
- BDB_EVENT_NO_NETWORK if no open network was discovered for joining.
- BDB_EVENT_NWK_JOIN_FAILURE if the node attempted to join a network but failed.

Network Steering is described in more detail in Network steering.

Parameters

- None

Returns

- BDB_E_SUCCESS

  (Network Steering has been successfully started)*.
- BDB_E_ERROR_IMPROPER_COMMISSIONING_MODE

  (Network Steering is not enabled).
- BDB_E_ERROR_COMMISSIONING_IN_PROGRESS

  (node is already in a commissioning mode).
- BDB_E_ERROR_INVALID_DEVICE

  (joining node is a Coordinator).

The eventual outcome is indicated by a *BDB_EVENT_NWK_STEERING_SUCCESS*, *BDB_EVENT_NO_NETWORK*, or *BDB_EVENT_NWK_JOIN_FAILURE* event, as described above.

Parent topic: Functions

BDB_eFbTriggerAsInitiator

```
BDB_teStatus BDB_eFbTriggerAsInitiator(
uint8 u8SourceEndPointId);
```

Description This function starts the Finding and Binding process on an initiator endpoint. The function may be called as the result of a user action, such as a button-press. The initiator remains in Finding and Binding mode for a fixed time-interval (in seconds) defined using the macro BDBC_MIN_COMMISSIONING_TIME in the **bdb_options.h** file.

The initiator node first searches for target endpoints by broadcasting an Identify Query command. If the initiator receives a response from a remote endpoint, it then sends a Simple Descriptor request to this endpoint. If the requested Simple Descriptor is successfully received back, then the initiator checks this descriptor for clusters on the remote endpoint that match its own clusters. If there is at least one matching cluster, the initiator does one of the following:

- If binding is required (indicated by the *u16bdbCommissioningGroupID* attribute being equal to 0xFFFF), the initiator adds the remote endpoint to the local Binding table.
- If grouping is required (indicated by the *u16bdbCommissioningGroupID* attribute being equal to a 16-bit group address), the initiator requests that the target endpoint adds the group address to its Group Address table.

Finding and Binding mode is described in Finding and Binding.

Note: Events are generated during this function call - for details, refer to Initiator Node.

Parameters

- *u8SourceEndPointId*: Number of initiator endpoints

Returns

- BDB_E_SUCCESS

  (Finding and Binding has been successfully started).
- BDB_E_FAILURE

  (invalid endpoint number or unable to broadcast Identify Query command).
- BDB_E_ERROR_COMMISSIONING_IN_PROGRESS

  (Finding and Binding already on-going).

Parent topic: Functions

BDB_vFbExitAsInitiator

```
void BDB_vFbExitAsInitiator(void);
```

Description This function stops an on-going Finding and Binding process on an initiator endpoint. The function may be called as the result of a user action, such as a button-press or button-release.

Finding and Binding mode is described in Finding and Binding.

Parameters

- None

Returns

- None

Parent topic: Functions

BDB_eFbTriggerAsTarget

```
BDB_teStatus BDB_eFbTriggerAsTarget(uint8 u8EndPoint);
```

Description This function starts the Finding and Binding process on a target endpoint and must be called locally by the application on the target endpoint. The function may be called as the result of a user action, such as a button-press.

The functions put the device into identification mode of the Identify cluster for a time-interval (in seconds) which is at least equal to the value defined using the macro BDBC_MIN_COMMISSIONING_TIME in the **bdb_options.h** file. During this time, the target device generates responses to Identify Query commands, as well as other Finding and Binding commands.

The endpoint is then brought out of Find and Binding mode locally using the function **BDB_vFbExitAsTarget()** or remotely (by the initiator) using the Identify cluster function **eCLD_IdentifyCommandIdentifyRequestSend()**.

Finding and Binding mode is described in Finding and Binding.

Parameters

- *u8EndPoint*: Number of target endpoints

Returns

- BDB_E_SUCCESS

  (Finding and Binding has been successfully started).
- BDB_E_FAILURE

  (invalid endpoint number or Identify cluster is inaccessible).

Parent topic: Functions

BDB_vFbExitAsTarget

```
void BDB_vFbExitAsTarget(uint8 u8SourceEndpoint);
```

Description This function stops an on-going Finding and Binding process on a target endpoint and must be called locally by the application on the target endpoint. The function may be called as the result of a user action, such as a button-press or button-release.

Finding and Binding mode is described in Section 2.2.4.

Parameters

- *u8SourceEndpoint*: Number of target endpoint

Returns

- None

Parent topic: Functions

BDB_bIsBaseIdle

```
bool_t BDB_bIsBaseIdle(void);
```

Description This function determines whether the ZigBee Base Device is busy or idle, and therefore whether the node can enter sleep mode. The function returns a Boolean indicating the activity status of the ZigBee Base Device.

If the ZigBee Base Device is idle and the node can go to sleep (indicated by TRUE), it is then the responsibility of the application to put the device into sleep mode.

Parameters

- None

Returns

- TRUE indicates that the ZigBee Base Device is idle and the node can sleep.
- FALSE indicates that the ZigBee Base Device is busy.

Parent topic: Functions

BDB_u8OutOfBandCommissionStartDevice

```
uint8 BDB_u8OutOfBandCommissionStartDevice(
BDB_tsOobWriteDataToCommission *psStartupData);
```

Description This function initiates out-of-band commissioning which allows the local device to form a network as a Coordinator or to join an existing network as a Router or End Device. The function should be called after **ZPS_eAplAfInit()**. It is called when commissioning data is received from another device via out-of-band means. This commissioning data must be supplied to the function in a BDB_tsOobWriteDataToCommission structure, described in Section 2.7.5.

Not all the data values are mandatory.

The out-of-band commissioning interface makes sensible assumptions about data values and does not allow certain values already in the node to be over-ridden by the commissioning data. For example:

- It does not allow the network address of a Coordinator to set to a non-zero value (since the network address of the Coordinator must be zero)
- It does not allow the rejoin flag to set on a Coordinator (since the Coordinator cannot leave and then rejoin the network)
- In a centralized network, it does not allow the IEEE/MAC address of Trust Centre to set to any value other than the IEEE/MAC address of Coordinator (since the Coordinator is always the Trust Centre)

For an overview of out-of-band commissioning, refer to Out-Of-Band Commissioning.

Parameters

- *psStartupData*: Pointer to a structure containing commissioning data (see BDB_tsOobWriteDataToCommission)

Returns

- BDB_E_SUCCESS

  (The device has successfully formed or joined a network).
- BDB_E_FAILURE

  (The request to form or join a network has not been accepted).
- ZPS_NWK_ENUM_INVALID_REQUEST

  (The request contained invalid data).
- ZPS_APL_APS_E_ILLEGAL_REQUEST

  (The stack is not in the correct state to accept the request).

Parent topic: Functions

BDB_vOutOfBandCommissionGetData

```
void BDB_vOutOfBandCommissionGetData(
BDB_tsOobReadDataToAuthenticate
*psReturnedCommissioningData);
```

Description This function is used to obtain locally stored commissioning data. The obtained data is received in a structure described in BDB_tsOobReadDataToAuthenticate and includes the network key. The data is then passed to higher layers which may encrypt it before sending it by out-of-band means to the other device involved in the commissioning.

A similar set of data but with the network key encrypted can be obtained using the function **BDB_eOutOfBandCommissionGetDataEncrypted()**.

For an overview of out-of-band commissioning, refer to Out-Of-Band Commissioning.

Parameters

- *psReturnedCommissioningData*: Pointer to a structure to receive the obtained commissioning data (see BDB_tsOobReadDataToAuthenticate)

Returns

- None

Parent topic: Functions

BDB_eOutOfBandCommissionGetDataEncrypted

```
BDB_teStatus BDB_eOutOfBandCommissionGetDataEncrypted(
BDB_tsOobWriteDataToAuthenticate *psSrcCredentials,
uint8 *pu8ReturnAuthData,
uint16 *puSize);
```

Description This function is used to obtain locally stored commissioning data, including the network key which returns encrypted. Authentication data (including an install code) must be provided which is used to encrypt the network key.

The obtained data is received as a byte stream - the size of the byte stream is also returned. The byte stream contains the following data:

- IEEE/MAC address of the local node as `u64address` (8 bytes)
- Network key encrypted with the data passed via *psSrcCredentials* (16 bytes)
- MIC value generated to validate encryption (4 bytes)
- Key sequence number of active network key (1 byte)
- Active channel number (1 byte)
- PAN ID (2 bytes)
- Extended PAN ID (8 bytes)

The encrypted network key and other obtained data are then sent by out-of-band means to the other device involved in the commissioning. The receiving device may decrypt the key using the **BDB_bOutOfBandCommissionGetKey()** function.

A similar set of data without encryption of the network key is obtained using the function **BDB_u8OutOfBandCommissionStartDevice()**.

For an overview of out-of-band commissioning, refer to Out-Of-Band Commissioning.

Parameters

- *psSrcCredentials*: Pointer to a structure containing authentication data to be used to encrypt the network key (see BDB_tsOobWriteDataToAuthenticate).
- *pu8ReturnAuthData*: Pointer to the start of the returned byte stream containing the obtained data.
- *puSize*: Pointer to a location to receive the size of the obtained byte stream.

Returns

- None

Parent topic: Functions

BDB_bOutOfBandCommissionGetKey

```
bool_tBDB_bOutOfBandCommissionGetKey(
uint8* pu8InstallCode,
uint8* pu8EncKey,
uint64 u64ExtAddress,
uint8* pu8DecKey,
uint8* pu8Mic);
```

Description This function is used to decrypt an encrypted security key. It may be used to decrypt the network key received from another device during out-of-band commissioning.

The function requires the install code that was used to generate the pre-configured link key used to encrypt the key.

For an overview of out-of-band commissioning, refer to Out-Of-Band Commissioning.

Parameters

- *pu8InstallCode*: Pointer to install code used to generate the pre-configured link key used in the encryption.
- *pu8EncKey*: Pointer to encrypted key.
- *u64ExtAddress*: Pointer to IEEE/MAC address of originating device.
- *pu8DecKey*: Pointer to location to receive the decrypted key.
- *pu8Mic*: Pointer to the MIC value to be used to validate the decryption.

Returns

- TRUE if key successfully decrypted, otherwise FALSE

Parent topic: Functions

Parent topic:*ZigBee Base Device*

Structures

BDB_tsBdbEvent The following structure contains ZigBee Base Device event information that is passed to the **APP_vBdbCallback()** callback function (see Events).

```
typedef struct
{
    BDB_teBdbEventTypee EventType;
    BDB_tuBdbEventData uEventData;
}BDB_tsBdbEvent;
```

where:

- eEventType is an enumeration indicating the event type - for the possible enumerations, refer to Section 2.9.
- uEventData is a union structure containing the event information (if any) - for a description of this structure, refer to Section 2.7.2.

Parent topic: Structures

BDB_tuBdbEventData The following structure is a union containing the data for a ZigBee Base Device event.

```
typedef union
{
BDB_tsZpsAfEvent sZpsAfEvent;
BDB_tsFindAndBindEvent *psFindAndBindEvent;
}BDB_tuBdbEventData
```

where:

- sZpsAfEvent is a structure containing the data for a stack event, indicated by the event type BDB_EVENT_ZPSAF - for a description of this structure, refer to Section 2.7.3.
- psFindAndBindEvent is a pointer to a structure containing the data for a 'Finding and Binding' event (see Section 2.9) - for a description of this structure, refer to Section 2.7.4.

Parent topic: Structures

BDB_tsZpsAfEvent The following structure contains the data for a ZigBee stack event (see the BDB_EVENT_ZPSAF event in Section 2.9).

```
typedef struct
{
    uint8 u8EndPoint;
    ZPS_tsAfEvent sStackEvent;
}BDB_tsZpsAfEvent;
```

where:

- u8EndPoint is the number of the endpoint on which the event occurred.
- sStackEvent is a ZPS structure containing the stack event type and data - this structure is detailed in the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

Parent topic: Structures

BDB_tsFindAndBindEvent The following structure contains the data for a 'Finding and Binding' event (see Section 2.9), which is passed to the application during the Finding and Binding process on the initiator.

```
typedef struct{
    uint8 u8InitiatorEp;
    uint8 u8TargetEp;
    uint16 u16TargetAddress;
    uint16 u16ProfileId;
    uint16 u16DeviceId;
    uint8 u8DeviceVersion;
    union {
        uint16 u16ClusterId;
        uint16 u16GroupId;
    }uEvent;
    ZPS_tsAfZdpEvent *psAfZdpEvent;
    bool bAllowBindOrGroup;
    bool bGroupCast;
}BDB_tsFindAndBindEvent;
```

where:

- u8InitiatorEp is the number of the endpoint involved in the binding/grouping on the initiator node.
- u8TargetEp is the number of the endpoint involved in the binding/grouping on the target node.
- u16TargetAddress is the 16-bit network address of the target node.
- u16ProfileId is the identifier of the ZigBee application profile supported by the two nodes (for Lighting & Occupancy devices, this is 0x0104)
- u16DeviceId is the 16-bit identifier of the ZigBee device type supported by the target endpoints. This must be a device type identifier issued by the ZigBee Alliance.
- u8DeviceVersion contains 4 bits (bits 0-3) representing the version of the supported device description on the target node (the default is 0000, unless set to another value according to the application profile used).
- uEvent is a union of the following two fields:
  - u16ClusterId is the identifier of the cluster involved in the binding.
  - u16GroupId is the address of the group to which the target endpoint is assigned.
- psAfZdpEvent is a pointer to a ZPS_tsAfZdpEvent structure containing the generated Finding and Binding event - this ZPS structure is detailed in the ZigBee 3.0 Stack User Guide (JNUG3130). The event can be any of the following (detailed in Section 2.9):
  - BDB_EVENT_FB_HANDLE_SIMPLE_DESC_RESP_OF_TARGET
  - BDB_EVENT_FB_CHECK_BEFORE_BINDING_CLUSTER_FOR_TARGET
  - BDB_EVENT_FB_CLUSTER_BIND_CREATED_FOR_TARGET
  - BDB_EVENT_FB_BIND_CREATED_FOR_TARGET
  - BDB_EVENT_FB_GROUP_ADDED_TO_TARGET
  - BDB_EVENT_FB_ERR_BINDING_FAILED
  - BDB_EVENT_FB_ERR_BINDING_TABLE_FULL
  - BDB_EVENT_FB_ERR_GROUPING_FAILED
  - BDB_EVENT_FB_NO_QUERY_RESPONSE
  - BDB_EVENT_FB_TIMEOUT
- bAllowBindOrGroup is a Boolean flag that indicates whether the relevant cluster is permitted to participate in a binding or grouping. The default value is TRUE (permitted) but if the

application must exclude the cluster (and block the binding/grouping) then it should set this field to FALSE.

- bGroupCast is a Boolean flag that indicates whether an 'Add Group If Identifying' command should be broadcast to all the identifying targets (TRUE) or an 'Add Group' request should be individually unicast to all the identifying targets. The default value is TRUE.

Parent topic: Structures

BDB_tsOobWriteDataToCommission The following structure contains the data values used to initialize a node at the start of out-of-band commissioning of the node.

```
typedef struct{
uint64 u64PanId;
uint64 u64TrustCenterAddress;
uint8* pu8NwkKey;
uint8* pu8InstallCode;
uint16 u16PanId;
uint16 u16ShortAddress;
bool_t bRejoin;
uint8 u8ActiveKeySqNum;
uint8 u8DeviceType;
uint8 u8RxOnWhenIdle;
uint8 u8Channel;
uint8 u8NwkUpdateId;
}BDB_tsOobWriteDataToCommission;
```

where:

- u64PanId is the Extended PAN ID of the network to be joined.
- u64TrustCenterAddress is the IEEE/MAC address of the Trust Centre in the centralized network to be joined.
- pu8NwkKey is a pointer to the network key.
- pu8InstallCode is a pointer to an initial link key derived from an install code (see Section 2.3.1).
- u16PanId is the PAN ID of the network to be joined.
- u16ShortAddress is the network address assigned to the node.
- bRejoin is the 'rejoin flag' which indicates whether the node should attempt to rejoin the network if it leaves (TRUE: rejoin, FALSE: do not rejoin).
- u8ActiveKeySqNum is the key sequence number associated with the active network key.
- u8DeviceType is a value indicating the type of ZigBee node:
  - 0: Coordinator
  - 1: Router
  - 2: End Device All other values are reserved.
- u8RxOnWhenIdle is a value indicating whether the receiver of the node is enabled during idle periods:
  - 0: Receiver off when idle (sleeping device)
  - 1: Receiver on when idle (non-sleeping device) All other values are reserved.
- u8Channel is the radio channel number on which the network operates.

- u8NwkUpdateId is a unique byte value which is incremented when the network parameters are updated (and is therefore used to determine whether a receiving node has missed an update).

Parent topic: Structures

BDB_tsOobReadDataToAuthenticate The following structure contains data values that are read from the local node during out-of-band commissioning of the node.

```
typedef struct{
uint8 au8Key[16]___attribute___((aligned(16)));
uint64 u64TcAddress;
uint64 u64PanId;
uint16 u16ShortPanId;
uint8 u8ActiveKeySeq;
uint8 u8Channel;
}BDB_tsOobReadDataToAuthenticate;
```

where:

- au8Key[16]___attribute___((aligned (16))) is an array containing the current network key, with one byte per array element.
- u64TcAddress is the IEEE/MAC address of the Trust Centre of the network to which the node is being commissioned.
- u64PanId is the Extended PAN ID of the network to which the node is being commissioned.
- u16ShortPanId is the PAN ID of the network to which the node is being commissioned.
- u8ActiveKeySeq is the key sequence number of the currently active network key.
- u8Channel is the radio channel number on which the network operates.

Parent topic: Structures

BDB_tsOobWriteDataToAuthenticate The following structure contains authentication data that is used to encrypt a security key during out-of-band commissioning of the node.

```
typedef struct{
    uint64 u64ExtAddr;
    uint8* pu8InstallCode;
}BDB_tsOobWriteDataToAuthenticate;
```

where:

- u64ExtAddr is the IEEE/MAC address of the node.
- pu8InstallCode is a pointer to a 16-bit install code to be used in the key encryption.

Parent topic: Structures

Parent topic:*ZigBee Base Device*

Enumerations This section lists and describes the enumerations used on the ZigBee Base Device. However, the ZigBee Base Device event enumerations are detailed in Events.

BDB_teStatus The following enumerations indicate the status of certain function calls.

```
typedef enum
{
    BDB_E_SUCCESS,
    BDB_E_FAILURE,
    BDB_E_ERROR_INVALID_PARAMETER,
    BDB_E_ERROR_INVALID_DEVICE,
    BDB_E_ERROR_NODE_IS_ON_A_NWK,
    BDB_E_ERROR_IMPROPER_COMMISSIONING_MODE,
    BDB_E_ERROR_COMMISSIONING_IN_PROGRESS,
}BDB_teStatus;
```

The enumerations are listed and described in the *Table 11*:

Enumeration	Description
BDB_E_SUCCESS	Function call is successful in its purpose
BDB_E_FAILURE	Function call fails in its purpose and no other error code is appropriate
BDB_E_ERROR_INVALID_PARAMETER	A specified parameter value is invalid
BDB_E_ERROR_INVALID_DEVICE	Device type is not valid for the operation
BDB_E_ERROR_NODE_IS_ON_A_NWK	Node is already in a network
BDB_E_ERROR_IMPROPER_COMMISSIO	The commissioning mode is not appropriate
BDB_E_ERROR_COMMISSIONING_IN_PI	The commissioning process is in progress

Parent topic: Enumerations

BDB_teCommissioningStatus The following enumerations are used to indicate the status of the commissioning process for the node.

```
typedef enum
{
    E_BDB_COMMISSIONING_STATUS_SUCCESS,
    E_BDB_COMMISSIONING_STATUS_IN_PROGRESS,
    E_BDB_COMMISSIONING_STATUS_NOT_AA_CAPABLE,
    E_BDB_COMMISSIONING_STATUS_NO_NETWORK,
    E_BDB_COMMISSIONING_STATUS_FORMATION_FAILURE,
    E_BDB_COMMISSIONING_STATUS_NO_IDENTIFY_QUERY_RESPONSE,
    E_BDB_COMMISSIONING_STATUS_BINDING_TABLE_FULL,
    E_BDB_COMMISSIONING_STATUS_NO_SCAN_RESPONSE,
    E_BDB_COMMISSIONING_STATUS_NOT_PERMITTED,
    E_BDB_COMMISSIONING_STATUS_TCLK_EX_FAILURE
}BDB_teCommissioningStatus;
```

The enumerations are listed and described in the table below:

Commissioning Status Enumerations

Enumeration	Description
E_BDB_COMMISSIONING_STATUS_SUCCESS	Commissioning is successfully completed
E_BDB_COMMISSIONING_STATUS_IN_PROGRESS	Commissioning is on-going
E_BDB_COMMISSIONING_STATUS_NOT_AA_CAPABL	Parent cannot assign address to joining node
E_BDB_COMMISSIONING_STATUS_NO_NETWORK	No network is found that can be joined
E_BDB_COMMISSIONING_STATUS_FORMATION_FAI	Network formation failed
E_BDB_COMMISSIONING_STATUS_NO_IDENTIFY_QU _ RESPONSE	No responses received to an Identify Query command
E_BDB_COMMISSIONING_STATUS_BINDING_TABLE_	The local Binding table is full
E_BDB_COMMISSIONING_STATUS_NO_SCAN_RESPO	No responses received during a channel scan
E_BDB_COMMISSIONING_STATUS_NOT_PERMITTED	Requested commissioning is not permitted
E_BDB_COMMISSIONING_STATUS_TCLK_EX_FAILUR	Trust Centre link key exchange failed

Parent topic: Enumerations

Parent topic:*ZigBee Base Device*

Events The ZigBee Base Device has a number of associated events. Some API functions (described in Functions) return immediately and the outcome of the process they invoke is later indicated with the generation of an asynchronous event. A user-defined callback function must be defined in the application to handle these events. The prototype of this callback function is as follows:

void APP_vBdbCallback(BDB_tsBdbEvent ****psBdbEvent*)**

where *psBdbEvent* is a pointer to a BDB_tsBdbEvent event structure containing the event information to be passed to the function (for this structure, see BDB_tsBdbEvent).

The enumerations for the ZigBee Base Device events are listed below.

```
typedef enum
 {
   BDB_EVENT_NONE,
   BDB_EVENT_ZPSAF,
   BDB_EVENT_INIT_SUCCESS,
   BDB_EVENT_REJOIN_SUCCESS,
   BDB_EVENT_REJOIN_FAILURE,
   BDB_EVENT_NWK_STEERING_SUCCESS,
   BDB_EVENT_NO_NETWORK,
   BDB_EVENT_NWK_JOIN_SUCCESS,
   BDB_EVENT_NWK_JOIN_FAILURE,
   BDB_EVENT_APP_START_POLLING,
   BDB_EVENT_NWK_FORMATION_SUCCESS,
   BDB_EVENT_NWK_FORMATION_FAILURE,
   BDB_EVENT_FB_HANDLE_SIMPLE_DESC_RESP_OF_TARGET,
   BDB_EVENT_FB_CHECK_BEFORE_BINDING_CLUSTER_FOR_TARGET,
   BDB_EVENT_FB_CLUSTER_BIND_CREATED_FOR_TARGET,
   BDB_EVENT_FB_BIND_CREATED_FOR_TARGET,
   BDB_EVENT_FB_GROUP_ADDED_TO_TARGET,
   BDB_EVENT_FB_ERR_BINDING_FAILED,
   BDB_EVENT_FB_ERR_BINDING_TABLE_FULL,
   BDB_EVENT_FB_ERR_GROUPING_FAILED,
   BDB_EVENT_FB_NO_QUERY_RESPONSE,
   BDB_EVENT_FB_TIMEOUT,
   BDB_EVENT_FB_OVER_AT_TARGET,
```

(continues on next page)

(continued from previous page)

```
  BDB_EVENT_LEAVE_WITHOUT_REJOIN,
}
BDB_teBdbEventType;
```

These events are described below.

The events with 'FB' in their names are used in the 'Finding and Binding' process and the event data is contained in the structure BDB_tsFindAndBindEvent (see BDB_tsFindAndBindEvent).

Note: In addition, certain ZCL events are generated during the Finding and Binding process. These events are passed to the callback function **BDB_vZclEventHandler()**, which is supplied with the ZigBee Base Device. For these events, refer to Finding and Binding.

1. **BDB_EVENT_ZPSAF**

   This event indicates that a ZigBee stack event has occurred. In this case, the uEventData field (of the BDB_tsBdbEvent structure) contains a BDB_tsZpsAfEvent structure, which itself includes the ZPS_tsAfEvent stack event structure.

2. **BDB_EVENT_INIT_SUCCESS**

   This event is generated when the ZigBee Base Device has been successfully initialized.

3. **BDB_EVENT_REJOIN_SUCCESS**

   This event is generated when the node has successfully rejoined its previous network.

4. **BDB_EVENT_REJOIN_FAILURE**

   This event is generated when the node attempts to rejoin its previous network has failed.

5. **BDB_EVENT_NWK_STEERING_SUCCESS**

   This event is generated when the Network Steering process has successfully completed and the local node has broadcast either of the following messages:

   - Management Permit Joining message to request the network to be opened for other devices to join (this message is broadcast when the local node was already in the network before Network Steering).
   - Device Announce message to announce that the local node has joined the network (this message is broadcast when the local node was not in the network before Network Steering).

6. **BDB_EVENT_NO_NETWORK**

   This event is generated when no open network open was discovered in a channel scan performed by a device attempting to join a network.

7. **BDB_EVENT_NWK_JOIN_SUCCESS**

   This event is generated when the node has successfully joined a network.

8. **BDB_EVENT_NWK_JOIN_FAILURE**

   This event is generated when the node attempted to join a network but failed.

9. **BDB_EVENT_APP_START_POLLING**

   This event is generated on an End Device during the Trust Centre link key exchange procedure to instruct the application to start fast polling of its parent, in order to retrieve packets received as part of the exchange procedure.

10. **BDB_EVENT_NWK_FORMATION_SUCCESS**

    This event is generated at the end of the Network Formation process when a centralized or distributed has been successfully formed by the local node.

11. **BDB_EVENT_NWK_FORMATION_FAILURE**

    This event is generated at the end of the Network Formation process if the local node failed to form a network.

12. **BDB_EVENT_FB_HANDLE_SIMPLE_DESC_RESP_OF_TARGET**

    This event indicates that the initiator has received a Simple Descriptor response from a target. This event can be used by the application to determine which type of device (for example: Dimmable Light, On/Off Light) the initiator is binding to. The information provided to the application is:

```
u8InitiatorEp
u8TargetEp
u16TargetAddress
u16ProfileId
u16DeviceId
u8DeviceVersion
psAfZdpEvent \(points to received Simple Descriptor\)
```

13. **BDB_EVENT_FB_CHECK_BEFORE_BINDING_CLUSTER_FOR_TARGET**

    This event is generated just before creating a Binding table entry for a cluster. It allows the application to exclude clusters from binding by setting the bAllowBindOrGroup flag to FALSE (by default it is TRUE). This event can also be used when the application must perform a group binding by setting the attribute *u16bdbCommissioningGroupID* to a value other than 0xFFFF. Moreover, this event also allows the application to decide whether to broadcast an 'Add Group If Identifying' or unicast an 'Add Group' command. An 'Add Group If Identifying' is broadcast to all the identifying targets by setting bGroupCast to TRUE. By default, this parameter is set to FALSE and allows unicasting an 'Add Group' request individually to all the identifying targets. The information provided to the application is:

```
u8InitiatorEp
u8TargetEp
u16TargetAddress
u16ClusterId
bAllowBindOrGroup
bGroupCast
psAfZdpEvent \(points to received Simple Descriptor\)
```

14. **BDB_EVENT_FB_CLUSTER_BIND_CREATED_FOR_TARGET**

    This event is generated per cluster for every binding or grouping created. The event may be generated more than once for the same target device. For example, when binding a color Dimmer Switch to a Dimmable Light, the event is generated twice: once for the On/Off cluster and once for the Level Control Cluster. The information provided to the application is:

```
u8InitiatorEp
u8TargetEp
u16TargetAddress
u16ClusterId
```

15. **BDB_EVENT_FB_BIND_CREATED_FOR_TARGET**

    This event is generated once all address bindings have been completed. The application can then send a 'Stop Identifying' command to the bound target. The information provided to the application is:

```
u8InitiatorEp
u8TargetEp
u16TargetAddress
```

16. **BDB_EVENT_FB_GROUP_ADDED_TO_TARGET**

This event is generated once the 'Add Group' or 'Add Group If Identifying' has been sent, in order to inform the application that grouping has been completed from the perspective of the initiator. The application can then groupcast a 'Stop Identifying' command to the grouped targets. The information provided to the application is:

```
u8InitiatorEp
u8TargetEp
u16GroupId
u16TargetAddress
psAfZdpEvent
```

17. **BDB_EVENT_FB_ERR_BINDING_FAILED**

    This event is generated to indicate that an unexpected error has occurred while creating a Binding table entry.

18. **BDB_EVENT_FB_ERR_BINDING_TABLE_FULL**

    This event is generated to inform the application that the Binding table is full and therefore the Finding and Binding process has failed. As a result, the ZigBee Base Device exits the Finding and Binding process.

19. **BDB_EVENT_FB_ERR_GROUPING_FAILED**

    This event is generated to indicate that a grouping has failed, since the initiator was not able to send an 'Add Group' or 'Add Group If Identifying' request.

20. **BDB_EVENT_FB_NO_QUERY_RESPONSE**

    This event indicates that the initiator did not receive an Identify Query response within BDB_FB_RESEND_IDENTIFY_QUERY_TIME (default value is 10) seconds. The information provided to the application is:

    ```
    u8InitiatorEp
    ```

21. **BDB_EVENT_FB_TIMEOUT**

    This event indicates that the commissioning timer expired after a period defined by the constant BDBC_MIN_COMMISSIONING_TIME (180 seconds by default). The information provided to the application is:

    ```
    u8InitiatorEp
    ```

22. **BDB_EVENT_FB_OVER_AT_TARGET**

    This event indicates that the Finding and Binding process has ended on the target node because the identify time reached zero or a remote node forced it to go to zero.

23. **BDB_EVENT_LEAVE_WITHOUT_REJOIN**

    This event is generated when the node has been instructed to leave the network without attempting to rejoin the network.

Parent topic:*ZigBee Base Device*

Compile-time Options Compile-time options can be configured through definitions in the file **bdb_options.h**. This option allows custom values to be defined for ZigBee Base Device attributes and constants. If the value of an attribute or constant is not defined in this file, the default value for the attribute or constant is used.

Attributes The following macros can be used to pre-configure values for the ZigBee Base Device attributes (listed and described in Section 2.5.1):

- BDB_COMMISSIONING_GROUP_ID

- BDB_COMMISSIONING_MODE
- BDB_COMMISSIONING_STATUS
- BDB_JOINING_NODE_EUI64
- BDB_JOIN_USES_INSTALL_CODE_KEY
- BDB_NODE_JOIN_LINK_KEY_TYPE
- BDB_PRIMARY_CHANNEL_SET
- BDB_SCAN_DURATION
- BDB_SECONDARY_CHANNEL_SET
- BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS
- BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS_MAX
- BDB_TC_LINK_KEY_EXCHANGE_METHOD
- BDB_TRUST_CENTER_NODE_JOIN_TIMEOUT
- BDB_TRUST_CENTER_REQUIRE_KEYEXCHANGE

For example, to set the maximum number of key establishment attempts to 5, include the following line:

```
#define BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS_MAX5
```

Constants The following macros can be used to set values for the ZigBee Base Device constants (listed and described in Section 2.5.2):

- BDBC_MAX_SAME_NETWORK_RETRY_ATTEMPTS
- BDBC_MIN_COMMISSIONING_TIME
- BDBC_REC_SAME_NETWORK_RETRY_ATTEMPTS
- BDBC_TC_LINK_KEY_EXCHANGE_TIMEOUT
- BDBC_TL_INTERPAN_TRANS_ID_LIFETIME
- BDBC_TL_MIN_STARTUP_DELAY_TIME
- BDBC_TL_PRIMARY_CHANNEL_SET
- BDBC_TL_RX_WINDOW_DURATION
- BDBC_TL_SCAN_TIME_BASE_DURATION_MS
- BDBC_TL_SECONDARY_CHANNEL_SET

For example, to set the minimum commissioning time for which a network is open to join to 240 seconds, include the following line:

```
#define BDBC_MIN_COMMISSIONING_TIME240
```

(this minimum commissioning time should set to a value below 255 seconds)

Parent topic:*ZigBee Base Device*

Lighting and Occupancy Device Types This chapter details the ZigBee device types that are collected together in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)* from the ZigBee Alliance.

Note: Lighting and Occupancy are not the application profiles. The devices in this collection use the application profile identifier 0x0104, previously used for the Home Automation application profile. This identifier ensures backward compatibility with applications for devices based on the Home Automation 1.2 profile.

The ZigBee Lighting and Occupancy (ZLO) device types are listed in the table below.

Lighting and Occupancy Device Types

Device Type	Device ID	Reference
On/Off Light	0x0100	Section 3.1
Dimmable Light	0x0101	Section 3.2
Colour Dimmable Light	0x0102	Section 3.3
On/Off Light Switch	0x0103	Section 3.4
Dimmer Switch	0x0104	Section 3.5
Colour Dimmer Switch	0x0105	Section 3.6
Light Sensor	0x0106	Section 3.7
Occupancy Sensor	0x0107	Section 3.8
On/Off Ballast	0x0108	Section 3.9
Dimmable Ballast	0x0109	Section 3.10
On/Off Plug-in Unit	0x010A	Section 3.11
Dimmable Plug-in Unit	0x010B	Section 3.12
Colour Temperature Light	0x010C	Section 3.13
Extended Colour Light	0x010D	Section 3.14
Light Level Sensor	0x010E	Section 3.15
Colour Controller	0x0800	Section 3.16
Colour Scene Controller	0x0810	Section 3.17
Non-Colour Controller	0x0820	Section 3.18
Non-Colour Scene Controller	0x0830	Section 3.19
Control Bridge	0x0840	Section 3.20
On/Off Sensor	0x0850	Section 3.21

On/Off light The On/Off Light device is simply a light that can be switched on and off (two states only and no intermediate levels).

- The Device ID is 0x0100.
- The header file for the device is **on_off_light.h**.
- The clusters supported by the device are listed in Section 3.1.1.
- The device structure, tsZLO_OnOffLightDevice, is listed in Section 3.1.2.
- The endpoint registration function for the device, **eZLO_RegisterOnOffLightEndPoint()**, is detailed in Section 3.1.3.

Supported clusters The clusters used by the On/Off Light device are listed in the table below.

Table 14. Clusters for On/Off Light

Server (Input) side	Client (Output) side
Mandatory	
Basic	
Identify	
On/Off	
Scenes	
Groups	
Optional	
Level control	OTA upgrade
Touchlink commissioning	Occupancy sensing

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

Parent topic: On/Off light

Device structure The following tsZLO_OnOffLightDevice structure is the shared structure for an On/Off Light device:

```
typedef struct
{
        tsZCL_EndPointDefinition sEndPoint;
 /* Cluster instances */
        tsZLO_OnOffLightDeviceClusterInstances sClusterInstance;
        #if (defined CLD_BASIC) && (defined BASIC_SERVER)
/* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
        #endif
        #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
/* On/Off Cluster - Server */
        tsCLD_OnOff sOnOffServerCluster;
        tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
        #endif
        #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
/* Groups Cluster - Server */
        tsCLD_Groups sGroupsServerCluster;
        tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
        #endif
        #if (defined CLD_SCENES) && (defined SCENES_SERVER)
/* Scenes Cluster - Server */
        tsCLD_Scenes sScenesServerCluster;
        tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
        #endif
        #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
/* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                sIdentifyServerCustomDataStructure;
        #endif
```

(continues on next page)

(continued from previous page)

```
/* On Off light device 2 optional clusters for the server */
        #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
/* LevelControl Cluster - Server */
        tsCLD_LevelControl sLevelControlServerCluster;
        tsCLD_LevelControlCustomDataStructure
                sLevelControlServerCustomDataStructure;
        #endif
        #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
        tsCLD_ZllCommission sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
        sZllCommissionServerCustomDataStructure;
        #endif
/* On Off light device 2 optional clusters for the client */
        #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
        #endif
        #if (defined CLD_OCCUPANCY_SENSING) && (defined
        OCCUPANCY_SENSING_CLIENT)
        /* Occupancy Sensing Cluster - Client */
        tsCLD_OccupancySensing sOccupancySensingClientCluster;
        #endif
} tsZLO_OnOffLightDevice;
```

Parent topic: On/Off light

Registration function The following **eZLO_RegisterOnOffLightEndPoint()** function is the endpoint registration function for an On/Off Light device.

```
teZCL_Status eZLO_RegisterOnOffLightEndPoint(
    uint8 u8EndPointIdentifier,
    CL_ZCLCallBackFunction cbCallBack,
      tsZLO_OnOffLightDevice *psDeviceInfo);
```

Description This function is used to register an endpoint that supports an On/Off Light device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_OnOffLightDevice structure, described in Section 3.1.2. This structure stores all variables relating to the colour Controller device associated with the endpoint. This function sets the sEndPoint and sClusterInstance fields of this structure and these fields must not be directly written to by the application.

The function is called multiple times if more than one endpoint is used - for example, if more than one On/Off Light device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the callback function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered. on this endpoint (see Section 3.1.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: On/Off light

Parent topic:*Lighting and Occupancy Device Types*

Dimmable Light The Dimmable Light device is a light that can have its luminance varied, and can be switched on and off. The permitted range of light levels is 0x01 to 0xFE.

- The Device ID is 0x0101.
- The header file for the device is **dimmable_light.h**.
- The clusters supported by the device are listed in Section 3.2.1.
- The device structure, tsZLO_DimmableLightDevice, is listed in Section 3.2.2.
- The endpoint registration function for the device, **eZLO_RegisterDimmableLightEndPoint()**, is detailed in Section 3.2.3.

Supported clusters The clusters used by the Dimmable Light device are listed in the table below.

Table 15. Clusters for Dimmable Light

Server (Input) side	Client (Output) side
Mandatory	
Basic	
Identify	
On/Off	
Level control	
Scenes	
Groups	
Optional	
Touchlink commissioning	OTA upgrade Occupancy sensing

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01).*

Parent topic: Dimmable Light

Device Structure The following tsZLO_DimmableLightDevice structure is the shared structure for a Dimmable Light device:

```
typedefstruct
{
tsZCL_EndPointDefinitionsEndPoint;
/*Clusterinstances*/
tsZLO_DimmableLightDeviceClusterInstancessClusterInstance;
#if(definedCLD_BASIC)&&(definedBASIC_SERVER)
/*BasicCluster-Server*/
tsCLD_BasicsBasicServerCluster;
#endif
#if(definedCLD_IDENTIFY)&&(definedIDENTIFY_SERVER)
/*IdentifyCluster-Server*/
tsCLD_IdentifysIdentifyServerCluster;
tsCLD_IdentifyCustomDataStructure
sIdentifyServerCustomDataStructure;
#endif
#if(definedCLD_ONOFF)&&(definedONOFF_SERVER)
/*On/OffCluster-Server*/
tsCLD_OnOffsOnOffServerCluster;
tsCLD_OnOffCustomDataStructuresOnOffServerCustomDataStructure;
#endif
#if(definedCLD_GROUPS)&&(definedGROUPS_SERVER)
/*GroupsCluster-Server*/
```

(continues on next page)

(continued from previous page)

```
tsCLD_GroupssGroupsServerCluster;
tsCLD_GroupsCustomDataStructuresGroupsServerCustomDataStructure;
#endif
#if(definedCLD_SCENES)&&(definedSCENES_SERVER)
/*ScenesCluster-Server*/
tsCLD_ScenessScenesServerCluster;
tsCLD_ScenesCustomDataStructuresScenesServerCustomDataStructure;
#endif
#if(definedCLD_LEVEL_CONTROL)&&(definedLEVEL_CONTROL_SERVER)
/*LevelControlCluster-Server*/
tsCLD_LevelControlsLevelControlServerCluster;
tsCLD_LevelControlCustomDataStructure
sLevelControlServerCustomDataStructure;
#endif
#if(definedCLD_ZLL_COMMISSION)&&(definedZLL_COMMISSION_SERVER)
tsCLD_ZllCommissionsZllCommissionServerCluster;
tsCLD_ZllCommissionCustomDataStructure
sZllCommissionServerCustomDataStructure;
#endif
#if(definedCLD_OTA)&&(definedOTA_CLIENT)
/*OTAcluster-Client*/
tsCLD_AS_OtasCLD_OTA;
tsOTA_CommonsCLD_OTA_CustomDataStruct;
#endif
#if(definedCLD_OCCUPANCY_SENSING)&&(definedOCCUPANCY_SENSING_CLIENT)
/*OccupancySensingCluster-Client*/
tsCLD_OccupancySensingsOccupancySensingClientCluster;
#endif
}tsZLO_DimmableLightDevice;
```

Parent topic: Dimmable Light

Registration Function The following **eZLO_RegisterDimmableLightEndPoint()** function is the endpoint registration function for a Dimmable Light device.

```
teZCL_Status eZLO_RegisterDimmableLightEndPoint(
            uint8 u8EndPointIdentifier,
    tfpZCL_ZCLCallBackFunction cbCallBack,
        tsZLO_DimmableLightDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports a Dimmable Light device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_DimmableLightDevice structure, described in Section 3.2.2. This structure stores all variables relating to the color Controller device associated with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is used - for example, if more than one Dimmable Light device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.2.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Dimmable Light

Parent topic:*Lighting and Occupancy Device Types*

Colour Dimmable Light The Colour Dimmable Light device is a multi-colour light that can have its hue, saturation and luminance varied, and can be switched on and off.

- The Device ID is 0x0102.
- The header file for the device is **colour_dimmable_light.h**.
- The clusters supported by the device are listed in Section 3.3.1.
- The device structure, tsZLO_ColourDimmableLightDevice, is listed in Section 3.3.2.

- The endpoint registration function for the device, **eZLO_RegisterColourDimmableLightEndPoint()**, is detailed in Section 3.3.3.

Supported clusters The clusters used by the Colour Dimmable Light device are listed in the table below.

Table 19. Clusters for Colour Dimmer Switch

Server (input) side	Client (output) side
Mandatory	
Basic	On/Off
Identify	Level control
	Colour control
	Identify
Optional	
On/Off switch configuration	OTA upgrade
	Scenes
	Groups

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01).*

Parent topic: Colour Dimmable Light

Device Structure The following tsZLO_ColourDimmableLightDevice structure is the shared structure for a Colour Dimmable Light device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;
/* Cluster instances */
    tsZLO_DimmableLightDeviceClusterInstances sClusterInstance;
    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
/* Basic Cluster - Server */
    tsCLD_Basic sBasicServerCluster;
    #endif
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
/* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    tsCLD_IdentifyCustomDataStructure
    sIdentifyServerCustomDataStructure;
    #endif
    #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
/* On/Off Cluster - Server */
    tsCLD_OnOff sOnOffServerCluster;
    tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
    #endif
    #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
/* Groups Cluster - Server */
    tsCLD_Groups sGroupsServerCluster;
    tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
```

(continues on next page)

(continued from previous page)

```
    #endif
    #if (defined CLD_SCENES) && (defined SCENES_SERVER)
/* Scenes Cluster - Server */
    tsCLD_Scenes sScenesServerCluster;
    tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
    #endif
    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
/* LevelControl Cluster - Server */
    tsCLD_LevelControl sLevelControlServerCluster;
    tsCLD_LevelControlCustomDataStructure
    sLevelControlServerCustomDataStructure;
    #endif
    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
    tsCLD_ZllCommission sZllCommissionServerCluster;
    tsCLD_ZllCommissionCustomDataStructure
    sZllCommissionServerCustomDataStructure;
    #endif
    #if (defined CLD_OTA) && (defined OTA_CLIENT)
/* OTA cluster - Client */
    tsCLD_AS_Ota sCLD_OTA;
    tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif
    #if (defined CLD_OCCUPANCY_SENSING) && (defined
OCCUPANCY_SENSING_CLIENT)
/* Occupancy Sensing Cluster - Client */
    tsCLD_OccupancySensing sOccupancySensingClientCluster;
    #endif
} tsZLO_DimmableLightDevice;
```

Parent topic: Colour Dimmable Light

Registration Function The following **eZLO_RegisterColourDimmableLightEndPoint()** function is the endpoint registration function for a color Dimmable Light device.

```
teZCL_Status eZLO_RegisterColourDimmableLightEndPoint(
                    uint8 u8EndPointIdentifier,
            tfpZCL_ZCLCallBackFunction cbCallBack,
        tsZLO_ColourDimmableLightDevice *psDeviceInfo);
```

Description This function is used to register an endpoint that supports a color Dimmable Light device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_ColourDimmableLightDevice structure, described in Section 3.3.2. This structure stores all variables relating to the color Controller device associated with

the endpoint. This function sets the sEndPoint and sClusterInstance fields of this structure and these fields must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one color Dimmable Light device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.3.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Colour Dimmable Light

Parent topic:*Lighting and Occupancy Device Types*

On/Off Light Switch The On/Off Light Switch device is used to switch a light device on and off by sending on, off and toggle commands to the target device.

- The Device ID is 0x0103.
- The header file for the device is **on_off_light_switch.h**.
- The clusters supported by the device are listed in Section 3.4.1.
- The device structure, tsZLO_OnOffLightSwitchDevice, is listed in Section 3.4.2.
- The endpoint registration function for the device, **eZLO_RegisterOnOffLightSwitchEndPoint()**, is detailed in Section 3.4.3.

Supported clusters The clusters used by the On/Off Light Switch device are listed in in the table below.

Table 17. Clusters for On/Off Light Switch

Server (Input) side	Client (Output) side
Mandatory	
Basic	On/Off
Identify	Identify
Optional	
On/Off switch configuration	OTA upgrade
	Scenes
	Groups

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01).*

Parent topic: On/Off Light Switch

Device Structure The following tsZLO_OnOffLightSwitchDevice structure is the shared structure for an On/Off Light Switch device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;
    /* Cluster instances */
    tsZLO_OnOffLightSwitchDeviceClusterInstances sClusterInstance;
    /* Mandatory server clusters */
    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
    /* Basic Cluster - Server */
    tsCLD_Basic sBasicServerCluster;
    #endif
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
    /* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    tsCLD_IdentifyCustomDataStructure
    sIdentifyServerCustomDataStructure;
    #endif
    /* Recommended Optional server */
    #if (defined CLD_OOSC) && (defined OOSC_SERVER)
        /* On/Off Switch Configuration Cluster - Server */
    tsCLD_OOSC sOOSCServerCluster;
    #endif
/* Mandatory client clusters */
    #if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
        tsCLD_OnOff sOnOffClientCluster;
    #endif
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
    /* Identify Cluster - Client */
    tsCLD_Identify sIdentifyClientCluster;
    tsCLD_IdentifyCustomDataStructure
        sIdentifyClientCustomDataStructure;
```

(continues on next page)

(continued from previous page)

```
    #endif
    #if (defined CLD_BASIC) && (defined BASIC_CLIENT)
    /* Basic Cluster - Client */
        tsCLD_Basic sBasicClientCluster;
    #endif
    /* Recommended Optional client clusters */
    #if (defined CLD_SCENES) && (defined SCENES_CLIENT)
        /* Scenes Cluster - Client */
        tsCLD_Scenes sScenesClientCluster;
        tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
    #endif
    #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
    /* Groups Cluster - Client */
        tsCLD_Groups sGroupsClientCluster;
        tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
    #endif
    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif
} tsZLO_OnOffLightSwitchDevice;
```

Parent topic: On/Off Light Switch

Registration Function The following **eZLO_RegisterOnOffLightSwitchEndPoint()** function is the endpoint registration function for an On/Off Light Switch device.

```
teZCL_Status eZLO_RegisterOnOffLightSwitchEndPoint(
        uint8 u8EndPointIdentifier,
        tfpZCL_ZCLCallBackFunction cbCallBack,
    tsZLO_OnOffLightSwitchDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports an On/Off Light Switch device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_OnOffLightSwitchDevice structure, described in Device Structure. This structure stores all variables relating to the color Controller device associated with the endpoint. This function sets the sEndPoint and sClusterInstance fields of this structure and these fields must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one On/Off Light Switch device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack:*Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Device Structure). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: On/Off Light Switch

Parent topic:*Lighting and Occupancy Device Types*

Dimmer Switch The Dimmer Switch device is used to control a characteristic of a light (for example, luminance) and to switch the light device on and off.

- The Device ID is 0x0104.
- The header file for the device is **dimmer_switch.h**.
- The clusters supported by the device are listed in Section 3.5.1.
- The device structure, tsZLO_DimmerSwitchDevice, is listed in Section 3.5.2.
- The endpoint registration function for the device, **eZLO_RegisterDimmerSwitchEndPoint()**, is detailed in Section 3.5.3.

Supported clusters The clusters used by the Dimmer Switch device are listed in the table below.

Table: Clusters for Dimmer Switch	
Server (input) side	Client (output) side
Mandatory	
Basic	On/Off
Identify	Identify
	Level control
Optional	
On/Off switch configuration	OTA upgrade
	Scenes
	Groups

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01).*

Parent topic: Dimmer Switch

Device Structure The following tsZLO_DimmerSwitchDevice structure is the shared structure for a Dimmer Switch device:

```
typedef struct
{
  tsZCL_EndPointDefinition sEndPoint;
/* Cluster instances */
    tsZLO_DimmerSwitchDeviceClusterInstances sClusterInstance;
/* Mandatory server clusters */
#if (defined CLD_BASIC) && (defined BASIC_SERVER)
  /* Basic Cluster - Server */
  tsCLD_Basic sBasicServerCluster;
#endif
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
/* Identify Cluster - Server */
  tsCLD_Identify sIdentifyServerCluster;
  tsCLD_IdentifyCustomDataStructure
  sIdentifyServerCustomDataStructure;
#endif
/* Optional server clusters */
#if (defined CLD_OOSC) && (defined OOSC_SERVER)
  /* On/Off Switch Configuration Cluster - Server */
  tsCLD_OOSC sOOSCServerCluster;
#endif
/* Mandatory client clusters */
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
  /* Identify Cluster - Client */
  tsCLD_Identify sIdentifyClientCluster;
  tsCLD_IdentifyCustomDataStructure
  sIdentifyClientCustomDataStructure;
#endif
```

(continues on next page)

(continued from previous page)

```
#if (defined CLD_BASIC) && (defined BASIC_CLIENT)
  /* Basic Cluster - Client */
  tsCLD_Basic sBasicClientCluster;
  #endif
#if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
  tsCLD_OnOff sOnOffClientCluster;
#endif
#if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
  /* Level Control Cluster - Client */
  tsCLD_LevelControl sLevelControlClientCluster;
  tsCLD_LevelControlCustomDataStructure
        sLevelControlClientCustomDataStructure;
#endif
/* Recommended Optional client clusters */
#if (defined CLD_SCENES) && (defined SCENES_CLIENT)
    /* Scenes Cluster - Client */
    tsCLD_Scenes sScenesClientCluster;
    tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
#endif
#if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
  /* Groups Cluster - Client */
  tsCLD_Groups sGroupsClientCluster;
  tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
#endif
#if (defined CLD_OTA) && (defined OTA_CLIENT)
  tsCLD_AS_Ota sCLD_OTA;
  tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif
} tsZLO_DimmerSwitchDevice;
```

Parent topic: Dimmer Switch

Registration Function The following **eZLO_RegisterDimmerSwitchEndPoint()** function is the endpoint registration function for a Dimmer Switch device.

```
teZCL_Status eZLO_RegisterDimmerSwitchEndPoint(
        uint8 u8EndPointIdentifier,
        tfpZCL_ZCLCallBackFunction cbCallBack,
        tsZLO_DimmerSwitchDevice *psDeviceInfo);
```

Description This function is used to register an endpoint that supports a Dimmer Switch device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for application.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_DimmerSwitchDevice structure, described in Section 3.5.2. This structure stores all variables relating to the color Controller device associated with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one Dimmer Switch device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.5.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Dimmer Switch

Parent topic:*Lighting and Occupancy Device Types*

Colour Dimmer Switch The Colour Dimmer Switch device controls the hue, saturation and luminance of a multi-colour light, and to switch the light device on and off.

- The Device ID is 0x0105.
- The header file for the device is **colour_dimmer_switch.h**.
- The clusters supported by the device are listed in Section 3.6.1.
- The device structure, tsZLO_ColourDimmerSwitchDevice, is listed in Section 3.6.2.

- The endpoint registration function for the device, **eZLO_RegisterColourDimmerSwitchEndPoint()**, is detailed in Section 3.6.3.

Supported clusters The clusters used by the Colour Dimmer Switch device are listed in the table below.

Table 19. Clusters for Colour Dimmer Switch

Server (input) side	Client (output) side
Mandatory	
Basic	On/Off
Identify	Level control
	Colour control
	Identify
Optional	
On/Off switch configuration	OTA upgrade
	Scenes
	Groups

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01).*

Parent topic: Colour Dimmer Switch

Device Structure The following tsZLO_ColourDimmerSwitchDevice structure is the shared structure for a Colour Dimmer Switch device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;
    /* Cluster instances */
    tsZLO_ColourDimmerSwitchDeviceClusterInstances sClusterInstance;
    /* Mandatory server clusters */
    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
    /* Basic Cluster - Server */
    tsCLD_Basic sBasicServerCluster;
    #endif
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
    /* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    tsCLD_IdentifyCustomDataStructure
    sIdentifyServerCustomDataStructure;
    #endif
    /* Optional server clusters */
    #if (defined CLD_OOSC) && (defined OOSC_SERVER)
        /* On/Off Switch Configuration Cluster - Server */
        tsCLD_OOSC sOOSCServerCluster;
    #endif
    /* Mandatory client clusters */
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
        /* Identify Cluster - Client */
```

(continues on next page)

(continued from previous page)

```
        tsCLD_Identify sIdentifyClientCluster;
        tsCLD_IdentifyCustomDataStructure
        sIdentifyClientCustomDataStructure;
    #endif
    #if (defined CLD_BASIC) && (defined BASIC_CLIENT)
    /* Basic Cluster - Server */
        tsCLD_Basic sBasicClientCluster;
        #endif
    #if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
    tsCLD_OnOff sOnOffClientCluster;
    #endif
    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
    /* Level Control Cluster - Client */
        tsCLD_LevelControl sLevelControlClientCluster;
        tsCLD_LevelControlCustomDataStructure
        sLevelControlClientCustomDataStructure;
     #endif
    #if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_CLIENT)
        /* Colour Control Cluster - Client */
        tsCLD_ColourControl sColourControlClientCluster;
        tsCLD_ColourControlCustomDataStructure
        sColourControlClientCustomDataStructure;
    #endif
    /*Recommended Optional client clusters */
    #if (defined CLD_SCENES) && (defined SCENES_CLIENT)
    /* Scenes Cluster - Client */
    tsCLD_Scenes sScenesClientCluster;
    tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
    #endif
    #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
    /* Groups Cluster - Client */
    tsCLD_Groups sGroupsClientCluster;
    tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
    #endif
    #if (defined CLD_OTA) && (defined OTA_CLIENT)
    tsCLD_AS_Ota sCLD_OTA;
    tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif
} tsZLO_ColourDimmerSwitchDevice;
```

Parent topic: Colour Dimmer Switch

Registration Function The following **eZLO_RegisterColourDimmerSwitchEndPoint()** function is the endpoint registration function for a color Dimmer Switch device.

```
teZCL_Status eZLO_RegisterColourDimmerSwitchEndPoint(
                uint8 u8EndPointIdentifier,
        tfpZCL_ZCLCallBackFunction cbCallBack,
            tsZLO_DimmerSwitchDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports a color Dimmer Switch device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file. This parameter represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

You must also provide a pointer to a tsZLO_ColourDimmerSwitchDevice structure, described in Device Structure. This structure stores all variables relating to the color Controller device associated with the endpoint. This function sets the sEndPoint and sClusterInstance fields of this structure and these fields must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one color Dimmer Switch device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Device Structure). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Colour Dimmer Switch

Parent topic:*Lighting and Occupancy Device Types*

Light Sensor The Light Sensor device reports the illumination level in an area.

- The Device ID is 0x0106.
- The header file for the device is **light_sensor.h**.
- The clusters supported by the device are listed in Section 3.7.1.
- The device structure, tsZLO_LightSensorDevice, is listed in Section 3.7.2.
- The endpoint registration function for the device, **eZLO_RegisterLightSensorEndPoint()**, is detailed in Section 3.7.3.

Supported clusters The clusters used by the Light Sensor device are listed in the table below.

Table 20. Clusters for Light Sensor

Server (input) side	Client (output) side
Mandatory	
Basic	Identify
Identify	
Illuminance measurement	
Optional	
	OTA upgrade
	Groups

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

Parent topic: Light Sensor

Device Structure The following tsZLO_LightSensorDevice structure is the shared structure for a Light Sensor device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;
    /* Cluster instances */
    tsZLO_LightSensorDeviceClusterInstances sClusterInstance;
    /* Mandatory server clusters */
    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
    /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
            sIdentifyServerCustomDataStructure;
    #endif
    #if (defined CLD_ILLUMINANCE_MEASUREMENT) && (defined
            ILLUMINANCE_MEASUREMENT_SERVER)
        /* Illuminance Measurement Cluster - Server */
```

(continues on next page)

(continued from previous page)

```
        tsCLD_IlluminanceMeasurement sIlluminanceMeasurementServerCluster;
    #endif
    /* Optional server clusters */
    #if (defined CLD_POLL_CONTROL) && (defined POLL_CONTROL_SERVER)
        tsCLD_PollControl sPollControlServerCluster;
        tsCLD_PollControlCustomDataStructure
        sPollControlServerCustomDataStructure;
    #endif
    /* Mandatory server clusters */
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
        /* Identify Cluster - Client */
        tsCLD_Identify sIdentifyClientCluster;
        tsCLD_IdentifyCustomDataStructure
            sIdentifyClientCustomDataStructure;
    #endif
    /* Recommended Optional client clusters */
    #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
        /* Groups Cluster - Client */
        tsCLD_Groups sGroupsClientCluster;
        tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
    #endif
    #if (defined CLD_OTA) && (defined OTA_CLIENT)
    tsCLD_AS_Ota sCLD_OTA;
    tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif
} tsZLO_LightSensorDevice;
```

Parent topic: Light Sensor

Registration Function The following **eZLO_RegisterLightSensorEndPoint()** function is the endpoint registration function for a Light Sensor device.

```
teZCL_Status eZLO_RegisterLightSensorEndPoint(
                uint8 u8EndPointIdentifier,
     tfpZCL_ZCLCallBackFunction cbCallBack,
         tsZLO_LightSensorDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports a Light Sensor device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
        (tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_LightSensorDevice structure, described in Section 3.7.2. This structure stores all variables relating to the color Controller device associated with the endpoint. This function sets the sEndPoint and sClusterInstance fields of this structure and these fields must not be directly written to by the application.

The function is called multiple times if more than one endpoint is used - for example, if more than one Light Sensor device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.7.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Light Sensor

Parent topic:*Lighting and Occupancy Device Types*

Occupancy Sensor The Occupancy Sensor device reports the presence (or not) of occupants in an area.

- The Device ID is 0x0107.
- The header file for the device is **occupancy_sensor.h**.
- The clusters supported by the device are listed in Section 3.8.1.
- The device structure, tsZLO_OccupancySensorDevice, is listed in Section 3.7.2.
- The endpoint registration function for the device, **eZLO_RegisterOccupancySensorEndPoint()**, is detailed in Section 3.7.3.

Supported clusters The clusters used by the Occupancy Sensor device are listed in the table below.

Clusters for Occupancy Sensor device

Server (input) side	Client (output) side
Mandatory	
Basic	Identify
Identify	
Occupancy sensing	
Optional	
	OTA upgrade
	Groups

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01).*

Parent topic: Occupancy Sensor

Device Structure The following tsZLO_OccupancySensorDevice structure is the shared structure for an Occupancy Sensor device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;
    /* Cluster instances */
    tsZLO_OccupancySensorDeviceClusterInstances sClusterInstance;
    /* Mandatory server clusters */
  #if (defined CLD_BASIC) && (defined BASIC_SERVER)
    /* Basic Cluster - Server */
    tsCLD_Basic sBasicServerCluster;
  #endif
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
    /* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    tsCLD_IdentifyCustomDataStructure sIdentifyServerCustomDataStructure;
    #endif
    #if (defined CLD_OCCUPANCY_SENSING) && (defined OCCUPANCY_SENSING_SERVER)
        /* Occupancy Sensing Cluster - Server */
        tsCLD_OccupancySensing sOccupancySensingServerCluster;
    #endif
        /* Optional server clusters */
    #if (defined CLD_POLL_CONTROL) && (defined POLL_CONTROL_SERVER)
        tsCLD_PollControl sPollControlServerCluster;
        tsCLD_PollControlCustomDataStructure
            sPollControlServerCustomDataStructure;
    #endif
        /* Mandatory client clusters */
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
            /* Identify Cluster - Client */
            tsCLD_Identify sIdentifyClientCluster;
            tsCLD_IdentifyCustomDataStructure sIdentifyClientCustomDataStructure;
    #endif
```

(continues on next page)

(continued from previous page)

```
    /* Recommended Optional client clusters */
  #if (defined CLD_POLL_CONTROL) && (defined POLL_CONTROL_CLIENT)
    tsCLD_PollControl sPollControlClientCluster;
    tsCLD_PollControlCustomDataStructure
    sPollControlClientCustomDataStructure;
  #endif
  #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
    /* Groups Cluster - Client */
    tsCLD_Groups sGroupsClientCluster;
    tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
  #endif
  #if (defined CLD_OTA) && (defined OTA_CLIENT)
    tsCLD_AS_Ota sCLD_OTA;
    tsOTA_Common sCLD_OTA_CustomDataStruct;
  #endif
} tsZLO_OccupancySensorDevice;;
```

Parent topic: Occupancy Sensor

Registration Function The following **eZLO_RegisterOccupancySensorEndPoint()** function is the endpoint registration function for an Occupancy Sensor device.

```
teZCL_Status eZLO_RegisterOccupancySensorEndPoint(
  uint8 u8EndPointIdentifier,
  tfpZCL_ZCLCallBackFunction cbCallBack,
    tsZLO_OccupancySensorDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports an Occupancy Sensor device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
  (tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_OccupancySensorDevice structure, described in Section 3.8.2. This structure stores all variables relating to the color Controller device associated with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one Occupancy Sensor device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.

- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.8.2). This function sets the sEndPoint and sClusterInstance fields of this structure and these fields must not be directly written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Occupancy Sensor

Parent topic:*Lighting and Occupancy Device Types*

On/Off Ballast The On/Off Ballast is a lighting device that can be switched on/off from a controller device, such as an On/Off Light Switch or an Occupancy Sensor.

- The Device ID is 0x0108.
- The header file for the device is **on_off_ballast.h**.
- The clusters supported by the device are listed in Section 3.9.1.
- The device structure, tsZLO_OnOffBallastDevice, is listed in Section 3.9.2.
- The endpoint registration function for the device, **eZLO_RegisterOnOffBallastEndPoint()**, is detailed in Section 3.9.3.

Supported clusters The clusters used by the On/Off Ballast device are listed in the table below.

Table 22. Clusters for On/Off Ballast

Server (Input) side	Client (Output) side
Mandatory	
Basic	
Power configuration	
Device temperature configuration	
Identify	
Groups	
Scenes	
On/Off	
Ballast configuration	
Optional	
Level control	OTA upgrade
Illuminance level sensing	Illuminance measurement
Touchlink commissioning	Illuminance level sensing
	Occupancy sensing

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

Parent topic: On/Off Ballast

Device Structure The following tsZLO_OnOffBallastDevice structure is the shared structure for an On/Off Ballast device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;
    /* Cluster instances */
    tsZLO_OnOffBallastDeviceClusterInstances sClusterInstance;
    /* Mandatory server clusters */
    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
            sIdentifyServerCustomDataStructure;
    #endif
    #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
    /* On/Off Cluster - Server */
      tsCLD_OnOff sOnOffServerCluster;
      tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
```

(continues on next page)

(continued from previous page)

```
  #endif
  #if (defined CLD_SCENES) && (defined SCENES_SERVER)
  /* Scenes Cluster - Server */
    tsCLD_Scenes sScenesServerCluster;
    tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
  #endif
  #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
  /* Groups Cluster - Server */
    tsCLD_Groups sGroupsServerCluster;
    tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
  #endif
  /* Optional server clusters */
  #if (defined CLD_POWER_CONFIGURATION) && (defined
POWER_CONFIGURATION_SERVER)
  /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
  #endif
  #if (defined CLD_DEVICE_TEMPERATURE_CONFIGURATION) && (defined
      DEVICE_TEMPERATURE_CONFIGURATION_SERVER)
      /* Device Temperature Configuration Cluster - Server */
      tsCLD_DeviceTemperatureConfiguration
          sDeviceTemperatureConfigurationServerCluster;
  #endif
  #if (defined CLD_BALLAST_CONFIGURATION) && (defined
    BALLAST_CONFIGURATION_SERVER)
    tsCLD_BallastConfiguration sBallastConfigurationServerCluster;
  #endif
 /* Recommended Optional server clusters */
  #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
  /* LevelControl Cluster - Server */
      tsCLD_LevelControl sLevelControlServerCluster;
      tsCLD_LevelControlCustomDataStructure
      sLevelControlServerCustomDataStructure;
  #endif
  #if (defined CLD_ILLUMINANCE_LEVEL_SENSING) && (defined
      ILLUMINANCE_LEVEL_SENSING_SERVER)
    tsCLD_IlluminanceLevelSensing
    sIlluminanceLevelSensingServerCluster;
  #endif
  #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
    tsCLD_ZllCommission sZllCommissionServerCluster;
    tsCLD_ZllCommissionCustomDataStructure
          sZllCommissionServerCustomDataStructure;
  #endif
  /*Recommended Optional client clusters */
  #if (defined CLD_OTA) && (defined OTA_CLIENT)
      /* OTA cluster - Client */
      tsCLD_AS_Ota sCLD_OTA;
      tsOTA_Common sCLD_OTA_CustomDataStruct;
  #endif
  #if (defined CLD_ILLUMINANCE_MEASUREMENT) && (defined
ILLUMINANCE_MEASUREMENT_CLIENT)
      /* Illuminance Measurement Cluster - Client */
       tsCLD_IlluminanceMeasurement sIlluminanceMeasurementClientCluster;
  #endif
  #if (defined CLD_ILLUMINANCE_LEVEL_SENSING) && (defined
 ILLUMINANCE_LEVEL_SENSING_CLIENT)
    tsCLD_IlluminanceLevelSensing
      sIlluminanceLevelSensingClientCluster;
  #endif
  #if (defined CLD_OCCUPANCY_SENSING) && (defined
```

(continues on next page)

(continued from previous page)

```
  OCCUPANCY_SENSING_CLIENT)
    /* Occupancy Sensing Cluster - Client */
      tsCLD_OccupancySensing sOccupancySensingClientCluster;
    #endif
} tsZLO_OnOffBallastDevice;
```

Parent topic: On/Off Ballast

Registration Function The following **eZLO_RegisterOnOffBallastEndPoint()** function is the endpoint registration function for an On/Off Ballast device.

```
teZCL_Status eZLO_RegisterOnOffBallastEndPoint(
uint8 u8EndPointIdentifier,
tfpZCL_ZCLCallBackFunction cbCallBack,
tsZLO_OnOffBallastDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports an On/Off Ballast device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_OnOffBallastDevice structure, described in Section 3.9.2. This structure stores all variables relating to the color Controller device associated with the endpoint. This function sets the sEndPoint and sClusterInstance fields of this structure and these fields must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one On/Off Ballast device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.9.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS

2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: On/Off Ballast

Parent topic:*Lighting and Occupancy Device Types*

Dimmable Ballast The Dimmable Ballast is a lighting device that switches on/off or have its level adjusted from a controller device, such as a Dimmer Switch, or simply be switched on/off from an Occupancy Sensor.

- The Device ID is 0x0109.
- The header file for the device is **dimmable_ballast.h**.
- The clusters supported by the device are listed in Section 3.10.1.
- The device structure, tsZLO_DimmableBallastDevice, is listed in Section 3.10.2.
- The endpoint registration function for the device, **eZLO_RegisterDimmableBallastEndPoint**, is detailed in Section 3.10.3.

Supported clusters The clusters used by the Dimmable Ballast device are listed in the table below.

Table 23. Clusters for Dimmable Ballast

Server (Input) side	Client (Output) side
Mandatory	
Basic	
Power configuration	
Device temperature configuration	
Identify	
Groups	
Scenes	
On/Off	
Level control	
Ballast configuration	
Optional	
Illuminance level sensing	OTA upgrade
Touchlink commissioning	Illuminance measurement
	Illuminance level sensing
	Occupancy sensing

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01).*

Parent topic: Dimmable Ballast

Device Structure The following tsZLO_DimmableBallastDevice structure is the shared structure for a Dimmable Ballast device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;
   /* Cluster instances */
   tsZLO_DimmableBallastDeviceClusterInstances sClusterInstance;
  /* Mandatory server clusters */
  #if (defined CLD_BASIC) && (defined BASIC_SERVER)
    /* Basic Cluster - Server */
    tsCLD_Basic sBasicServerCluster;
  #endif
  #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
    /* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    tsCLD_IdentifyCustomDataStructure
```

(continues on next page)

(continued from previous page)

```
        sIdentifyServerCustomDataStructure;
  #endif
  #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
    /* On/Off Cluster - Server */
    tsCLD_OnOff sOnOffServerCluster;
    tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
  #endif
  #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
    /* LevelControl Cluster - Server */
    tsCLD_LevelControl sLevelControlServerCluster;
    tsCLD_LevelControlCustomDataStructure
    sLevelControlServerCustomDataStructure;
  #endif
#if (defined CLD_SCENES) && (defined SCENES_SERVER)
  /* Scenes Cluster - Server */
    tsCLD_Scenes sScenesServerCluster;
    tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
  #endif
  #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
  /* Groups Cluster - Server */
    tsCLD_Groups sGroupsServerCluster;
    tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
  #endif
  #if (defined CLD_POWER_CONFIGURATION) && (defined
 POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
  #endif
    #if (defined CLD_DEVICE_TEMPERATURE_CONFIGURATION) && (defined
  DEVICE_TEMPERATURE_CONFIGURATION_SERVER)
    /* Device Temperature Configuration Cluster - Server */
    tsCLD_DeviceTemperatureConfiguration
        sDeviceTemperatureConfigurationServerCluster;
    #endif
    #if (defined CLD_BALLAST_CONFIGURATION) && (defined
  BALLAST_CONFIGURATION_SERVER)
    tsCLD_BallastConfiguration sBallastConfigurationServerCluster;
    #endif
  /* Recommended Optional server clusters */
  #if (defined CLD_ILLUMINANCE_LEVEL_SENSING) && (defined
  ILLUMINANCE_LEVEL_SENSING_SERVER)
    tsCLD_IlluminanceLevelSensing
      sIlluminanceLevelSensingServerCluster;
  #endif
  #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
    tsCLD_ZllCommission sZllCommissionServerCluster;
    tsCLD_ZllCommissionCustomDataStructure
      sZllCommissionServerCustomDataStructure;
  #endif
/*Recommended Optional client clusters */
  #if (defined CLD_OTA) && (defined OTA_CLIENT)
    /* OTA cluster - Client */
    tsCLD_AS_Ota sCLD_OTA;
    tsOTA_Common sCLD_OTA_CustomDataStruct;
  #endif
  #if (defined CLD_ILLUMINANCE_MEASUREMENT) && (defined
  ILLUMINANCE_MEASUREMENT_CLIENT)
    /* Illuminance Measurement Cluster - Client */
    tsCLD_IlluminanceMeasurement sIlluminanceMeasurementClientCluster;
  #endif
    #if (defined CLD_ILLUMINANCE_LEVEL_SENSING) && (defined
```

(continues on next page)

(continued from previous page)

```
  ILLUMINANCE_LEVEL_SENSING_CLIENT)
    tsCLD_IlluminanceLevelSensing
      sIlluminanceLevelSensingClientCluster;
  #endif
  #if (defined CLD_OCCUPANCY_SENSING) && (defined
  OCCUPANCY_SENSING_CLIENT)
      /* Occupancy Sensing Cluster - Client */
    tsCLD_OccupancySensing sOccupancySensingClientCluster;
  #endif
} tsZLO_DimmableBallastDevice;
```

Parent topic: Dimmable Ballast

Registration Function The following **eZLO_RegisterDimmableBallastEndPoint()** function is the endpoint registration function for a Dimmable Ballast device.

```
teZCL_Status eZLO_RegisterDimmableBallastEndPoint(
uint8 u8EndPointIdentifier,
tfpZCL_ZCLCallBackFunction cbCallBack,
tsZLO_DimmableBallastDevice *psDeviceInfo);
```

Description This function is used to register an endpoint that supports a Dimmable Ballast device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_DimmableBallastDevice structure, described in Section 3.10.2. This structure stores all variables relating to the color Controller device associated with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application. The function is called multiple times if more than one endpoint is being used - for example, if more than one Dimmable Ballast device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.10.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Dimmable Ballast

Parent topic:*Lighting and Occupancy Device Types*

On/Off Plug-in Unit The On/Off Plug-in Unit device is typically used in nodes that contain a controllable mains plug or adaptor which includes an on/off switch. It may be controlled from a controller device such as an On/Off Light Switch.

- The Device ID is 0x010A.
- The header file for the device is **on_off_plug.h**.
- The clusters supported by the device are listed in Section 3.11.1.
- The device structure, tsZLO_OnOffPlugDevice, is listed in Section 3.11.2.
- The endpoint registration function for the device, **eZLO_RegisterOnOffPlugEndPoint()**, is detailed in Section 3.11.3.

Supported clusters The clusters supported by the On/Off Plug-in Unit device are listed in the table below. |

Table 24. Clusters for On/Off Plug-in Unit

Server (Input) side	Client (Output) side
Mandatory	
Basic	
Identify	
Groups	
Scenes	
On/Off	
Optional	
Level control	OTA upgrade

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01).*

Parent topic: On/Off Plug-in Unit

Device Structure The following tsZLO_OnOffPlugDevice structure is the shared structure for an On/Off Plug-in Unit device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;
  /* Cluster instances */
    tsZLO_OnOffPlugDeviceClusterInstances sClusterInstance;
  #if (defined CLD_BASIC) && (defined BASIC_SERVER)
    /* Basic Cluster - Server */
    tsCLD_Basic sBasicServerCluster;
  #endif
  #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
    /* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    tsCLD_IdentifyCustomDataStructure
        sIdentifyServerCustomDataStructure;
  #endif
  #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
    /* On/Off Cluster - Server */
    tsCLD_OnOff sOnOffServerCluster;
    tsCLD_OnOffCustomDataStructure
        sOnOffServerCustomDataStructure;
  #endif
  #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
  /* Groups Cluster - Server */
    tsCLD_Groups sGroupsServerCluster;
    tsCLD_GroupsCustomDataStructure
        sGroupsServerCustomDataStructure;
  #endif
  #if (defined CLD_SCENES) && (defined SCENES_SERVER)
    /* Scenes Cluster - Server */
```

(continues on next page)

(continued from previous page)

```
  tsCLD_Scenes sScenesServerCluster;
  tsCLD_ScenesCustomDataStructure
      sScenesServerCustomDataStructure;
#endif
#if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
  /* LevelControl Cluster - Server */
  tsCLD_LevelControl sLevelControlServerCluster;
  tsCLD_LevelControlCustomDataStructure
      sLevelControlServerCustomDataStructure;
#endif
#if (defined CLD_OTA) && (defined OTA_CLIENT)
  /* OTA cluster - Client */
  tsCLD_AS_Ota sCLD_OTA;
  tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif
} tsZLO_OnOffPlugDevice;
```

Parent topic: On/Off Plug-in Unit

Registration Function The following **eZLO_RegisterOnOffPlugEndPoint()** function is the endpoint registration function for an On/Off Plug-in Unit device.

```
teZCL_Status eZLO_RegisterOnOffPlugEndPoint(
uint8 u8EndPointIdentifier,
tfpZCL_ZCLCallBackFunction cbCallBack,
tsZLO_OnOffPlugDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports an On/Off Plug-in Unit device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_OnOffPlugDevice structure, described in Section 3.11.2. This structure stores all variables relating to the color Controller device associated with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application. The function is called multiple times if more than one endpoint is being used - for example, if more than one On/Off Plug-in Unit device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.

- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.11.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: On/Off Plug-in Unit

Parent topic:*Lighting and Occupancy Device Types*

Dimmable Plug-in Unit The Dimmable Plug-in Unit device is typically used in nodes that contain a controllable mains plug or adaptor which includes an adjustable output (to a lamp). It may be controlled from a controller device such as a Dimmer Switch or a Non-colour Controller.

- The Device ID is 0x010B.
- The header file for the device is **dimmable_plug.h**.
- The clusters supported by the device are listed in Section 3.12.1.
- The device structure, tsZLO_DimmablePlugDevice, is listed in Section 3.12.2.
- The endpoint registration function for the device, **eZLO_RegisterDimmablePlugEndPoint()**, is detailed in Section 3.12.3.

Supported clusters The clusters supported by the Dimmable Plug-in Unit device are listed in the table below.

Table 25. Clusters for Dimmable Plug-in Unit	
Server (Input) side	**Client (Output) side**
Mandatory	
Basic	
Identify	
Groups	
Scenes	
On/Off	
Level control	
Optional	
	OTA upgrade
The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.	

Parent topic: Dimmable Plug-in Unit

Device Structure The following tsZLO_DimmablePlugDevice structure is the shared structure for a Dimmable Plug-in Unit device:

```
typedefstruct
{
  tsZCL_EndPointDefinitionsEndPoint;
   /* Cluster instances */
   tsZLO_DimmablePlugDeviceClusterInstances sClusterInstance;
   #if (defined CLD_BASIC) && (defined BASIC_SERVER)
       /* Basic Cluster - Server */
       tsCLD_Basic sBasicServerCluster;
   #endif
   #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
   /* Identify Cluster - Server */
   tsCLD_Identify sIdentifyServerCluster;
   tsCLD_IdentifyCustomDataStructure
           sIdentifyServerCustomDataStructure;
   #endif
   #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
   /* On/Off Cluster - Server */
   tsCLD_OnOff sOnOffServerCluster;
   tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
   #endif
   #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
       /* Groups Cluster - Server */
       tsCLD_Groups sGroupsServerCluster;
       tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
   #endif
   #if (defined CLD_SCENES) && (defined SCENES_SERVER)
```

(continues on next page)

(continued from previous page)

```
    /* Scenes Cluster - Server */
    tsCLD_Scenes sScenesServerCluster;
    tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
    #endif
    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
    /* LevelControl Cluster - Server */
    tsCLD_LevelControl sLevelControlServerCluster;
    tsCLD_LevelControlCustomDataStructure
              sLevelControlServerCustomDataStructure;
    #endif
    #if (defined CLD_OTA) && (defined OTA_CLIENT)
    /* OTA cluster - Client */
    tsCLD_AS_Ota sCLD_OTA;
    tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif
} tsZLO_DimmablePlugDevice;
```

Parent topic: Dimmable Plug-in Unit

Registration Function The following **eZLO_RegisterDimmablePlugEndPoint()** function is the endpoint registration function for a Dimmable Plug-in Unit device.

```
teZCL_Status eZLO_RegisterDimmablePlugEndPoint(
        uint8 u8EndPointIdentifier,
        tfpZCL_ZCLCallBackFunction cbCallBack,
        tsZLO_DimmablePlugDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports a Dimmable Plug-in Unit device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_DimmablePlugDevice structure, described in Device Structure. This structure stores all variables relating to the color Controller device associated with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one Dimmable Plug-in Unit device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.

- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Device Structure). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Dimmable Plug-in Unit

Parent topic:*Lighting and Occupancy Device Types*

Colour Temperature Light The Colour Temperature Light device is typically used in nodes that contain a colour lamp with adjustable colour (and brightness) which operates using colour temperature.

- The Device ID is 0x010C.
- The header file for the device is **colour_temperature_light.h**.
- The clusters supported by the device are listed in Supported clusters.
- The device structure, tsZLO_ColourTemperatureLightDevice, is listed in Device Structure.
- The endpoint registration function for the device, **eZLO_RegisterColourTemperatureLightEndPoint()**, is detailed in Registration Function.

Supported clusters The clusters supported by the Colour Temperature Light device are listed in the table below.

Table 26. Clusters for Colour Temperature Light

Server (Input) side	Client (Output) side
Mandatory	
Basic	
Identify	
Groups	
Scenes	
On/Off	
Level control	
Colour control	
Optional	
Touchlink commissioning	OTA upgrade

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01).*

Parent topic: Colour Temperature Light

Device Structure The following tsZLO_ColourTemperatureLightDevice structure is the shared structure for a Colour Temperature Light device:

```
typedef struct
{
  tsZCL_EndPointDefinition sEndPoint;
  /* Cluster instances */
    tsZLO_ColourTemperatureLightDeviceClusterInstances sClusterInstance;
  #if (defined CLD_BASIC) && (defined BASIC_SERVER)
    /* Basic Cluster - Server */
    tsCLD_Basic sBasicServerCluster;
  #endif
  #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
  /* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    tsCLD_IdentifyCustomDataStructure
          sIdentifyServerCustomDataStructure;
  #endif
  #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
    /* On/Off Cluster - Server */
    tsCLD_OnOff sOnOffServerCluster;
    tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
  #endif
  #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
    /* Groups Cluster - Server */
    tsCLD_Groups sGroupsServerCluster;
    tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
  #endif
  #if (defined CLD_SCENES) && (defined SCENES_SERVER)
```

(continues on next page)

(continued from previous page)

```
    /* Scenes Cluster - Server */
    tsCLD_Scenes sScenesServerCluster;
    tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
  #endif
  #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
    /* LevelControl Cluster - Server */
    tsCLD_LevelControl sLevelControlServerCluster;
    tsCLD_LevelControlCustomDataStructure
    sLevelControlServerCustomDataStructure;
    #endif
  #if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_SERVER)
    /* Colour Control Cluster - Server */
    tsCLD_ColourControl sColourControlServerCluster;
    tsCLD_ColourControlCustomDataStructure
        sColourControlServerCustomDataStructure;
  #endif
  #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
    tsCLD_ZllCommission sZllCommissionServerCluster;
    tsCLD_ZllCommissionCustomDataStructure
    sZllCommissionServerCustomDataStructure;
  #endif
  #if (defined CLD_OTA) && (defined OTA_CLIENT)
    /* OTA cluster - Client */
    tsCLD_AS_Ota sCLD_OTA;
    tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif
} tsZLO_ColourTemperatureLightDevice;
```

Parent topic: Colour Temperature Light

Registration Function The following **eZLO_RegisterColourTemperatureLightEndPoint()** function is the endpoint registration function for a color Temperature Light device.

```
teZCL_Status eZLO_RegisterColourTemperatureLightEndPoint(
                uint8 u8EndPointIdentifier,
     tfpZCL_ZCLCallBackFunction cbCallBack,
        tsZLO_ColourTemperatureLightDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports a color Temperature Light device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_ColourTemperatureLightDevice structure, described in Device Structure. This structure stores all variables relating to the color Controller device associated

with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one color Temperature Light device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Device Structure). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Colour Temperature Light

Parent topic:*Lighting and Occupancy Device Types*

Extended Colour Light The Extended Colour Light device is typically used in nodes that contain a colour lamp with adjustable colour and brightness. This device supports a range of colour parameters, including hue/saturation, enhanced hue, colour temperature, colour loop, and XY.

- The Device ID is 0x010D.
- The header file for the device is **extended_colour_light.h**.
- The clusters supported by the device are listed in Section 3.14.1.
- The device structure, tsZLO_ExtendedColourLightDevice, is listed in Section 3.14.2.

- The endpoint registration function for the device, **eZLO_RegisterExtendedColourLightEndPoint()**, is detailed in Section 3.14.3.

Supported clusters The clusters supported by the Extended Colour Light device are listed in the table below.

Table 27. Clusters for Extended Colour Light

Server (Input) side	Client (Output) side
Mandatory	
Basic	
Identify	
Groups	
Scenes	
On/Off	
Level control	
Colour control	
Optional	
Touchlink commissioning	OTA upgrade

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

Parent topic: Extended Colour Light

Device Structure The following tsZLO_ExtendedColourLightDevice structure is the shared structure for an Extended color Light device:

```
typedef struct
{
  tsZCL_EndPointDefinition sEndPoint;
  /*Clusterinstances*/
    tsZLO_ExtendedColourLightDeviceClusterInstances
                                sClusterInstance;
  #if (defined CLD_BASIC) && (defined BASIC_SERVER)
    /* Basic Cluster - Server */
    tsCLD_Basic sBasicServerCluster;
  #endif
  #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
    /* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    tsCLD_IdentifyCustomDataStructure
            sIdentifyServerCustomDataStructure;
  #endif
  #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
```

(continues on next page)

(continued from previous page)

```
    /* On/Off Cluster - Server */
    tsCLD_OnOff sOnOffServerCluster;
    tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
  #endif
  #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
    /* Groups Cluster - Server */
    tsCLD_Groups sGroupsServerCluster;
    tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
    #endif
  #if (defined CLD_SCENES) && (defined SCENES_SERVER)
    /* Scenes Cluster - Server */
    tsCLD_Scenes sScenesServerCluster;
    tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
  #endif
  #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
    /* LevelControl Cluster - Server */
    tsCLD_LevelControl sLevelControlServerCluster;
    tsCLD_LevelControlCustomDataStructure
    sLevelControlServerCustomDataStructure;
  #endif
  #if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_SERVER)
    /* Colour Control Cluster - Server */
    tsCLD_ColourControl sColourControlServerCluster;
    tsCLD_ColourControlCustomDataStructure
    sColourControlServerCustomDataStructure;
  #endif
  #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
    tsCLD_ZllCommission sZllCommissionServerCluster;
    tsCLD_ZllCommissionCustomDataStructure
    sZllCommissionServerCustomDataStructure;
  #endif
  #if (defined CLD_OTA) && (defined OTA_CLIENT)
    /* OTA cluster - Client */
    tsCLD_AS_Ota sCLD_OTA;
    tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif
} tsZLO_ExtendedColourLightDevice;
```

Parent topic: Extended Colour Light

Registration Function The following **eZLO_RegisterExtendedColourLightEndPoint()** function is the endpoint registration function for an Extended color Light device.

```
teZCL_Status eZLO_RegisterExtendedColourLightEndPoint(
uint8 u8EndPointIdentifier,
tfpZCL_ZCLCallBackFunction cbCallBack,
tsZLO_ExtendedColourLightDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports an Extended color Light device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when

an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_ExtendedColourLightDevice structure, described in Section 3.14.2. This structure stores all variables relating to the color Controller device associated with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one Extended color Light device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.14.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Extended Colour Light

Parent topic:*Lighting and Occupancy Device Types*

Light Level Sensor The Light Level Sensor device measures the illumination level in an area. It can be used to switch on/off a lighting device, such as an On/Off Ballast.

- The Device ID is 0x010E.
- The header file for the device is **light_level_sensor.h**.
- The clusters supported by the device are listed in Section 3.15.1.
- The device structure, tsZLO_LightLevelSensorDevice, is listed in Section 3.15.2.
- The endpoint registration function for the device, **eZLO_RegisterLightLevelSensorEndPoint()**, is detailed in Section 3.15.3.

Supported clusters The clusters used by the Light Level Sensor device are listed in the table below.

Table 20. Clusters for Light Sensor

Server (input) side	Client (output) side
Mandatory	
Basic	Identify
Identify	
Illuminance measurement	
Optional	
	OTA upgrade
	Groups
Parent topic: Light Level Sensor	

Device Structure The following tsZLO_LightLevelSensorDevice structure is the shared structure for a Light Level Sensor device:

```
typedef struct
{
  tsZCL_EndPointDefinition sEndPoint;
 /* Cluster instances */
  tsZLO_LightLevelSensorDeviceClusterInstances sClusterInstance;
  /* Mandatory server clusters */
  #if (defined CLD_BASIC) && (defined BASIC_SERVER)
  /* Basic Cluster - Server */
  tsCLD_Basic sBasicServerCluster;
  #endif
  #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
    /* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    tsCLD_IdentifyCustomDataStructure
        sIdentifyServerCustomDataStructure;
  #endif
  #if (defined CLD_ILLUMINANCE_LEVEL_SENSING) && (defined
```

(continues on next page)

(continued from previous page)

```
ILLUMINANCE_LEVEL_SENSING_SERVER)
  tsCLD_IlluminanceLevelSensing
    sIlluminanceLevelSensingServerCluster;
#endif
/* Optional server clusters */
#if (defined CLD_POLL_CONTROL) && (defined POLL_CONTROL_SERVER)
  tsCLD_PollControl sPollControlServerCluster;
  tsCLD_PollControlCustomDataStructure
  sPollControlServerCustomDataStructure;
#endif
/* Mandatory server clusters */
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
  /* Identify Cluster - Client */
  tsCLD_Identify sIdentifyClientCluster;
  tsCLD_IdentifyCustomDataStructure
  sIdentifyClientCustomDataStructure;
#endif
/* Recommended Optional client clusters */
#if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
  /* Groups Cluster - Client */
  tsCLD_Groups sGroupsClientCluster;
  tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
#endif
#if (defined CLD_OTA) && (defined OTA_CLIENT)
  tsCLD_AS_Ota sCLD_OTA;
  tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif
} tsZLO_LightLevelSensorDevice;
```

Parent topic: Light Level Sensor

Registration Function The following **eZLO_RegisterLightLevelSensorEndPoint()** function is the endpoint registration function for a Light Level Sensor device.

```
teZCL_Status eZLO_RegisterLightLevelSensorEndPoint(
    uint8 u8EndPointIdentifier,
    tfpZCL_ZCLCallBackFunction cbCallBack,
        tsZLO_LightLevelSensorDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports a Light Level Sensor device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void( *tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_ColourControllerDevice structure, described in Section 3.16.2. This structure stores all variables relating to the color Controller device associated with the end-

point. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one Light Level Sensor device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.16.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Light Level Sensor

Parent topic:*Lighting and Occupancy Device Types*

Colour Controller The Colour Controller device is used in a node that issues colour-control commands to adjust the intensity or colour of a lighting device, or switch it on/off.

- The Device ID is 0x0800.
- The header file for the device is **colour_controller.h**.
- The clusters supported by the device are listed in Section 3.16.1.
- The device structure, tsZLO_ColourControllerDevice, is listed in Section 3.16.2.

- The endpoint registration function for the device, **eZLO_RegisterColourRemoteEndPoint()**, is detailed in Section 3.16.3.

Supported clusters The clusters supported by the Colour Controller device are listed in the table below.

Table 29. Clusters for Colour Controller

Server (Input) side	Client (Output) side
Mandatory	
Basic	On/Off
Identify	Identify
	Level control
	Colour control
Optional	
Touchlink commissioning	Touchlink commissioning
	Groups
	OTA upgrade

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01).*

Parent topic: Colour Controller

Device Structure The following tsZLO_ColourControllerDevice structure is the shared structure for a Colour Controller device:

```
typedef struct
{
tsZCL_EndPointDefinitionsEndPoint;
/* Clusterinstances */
tsZLO_ColourControllerDeviceClusterInstances sClusterInstance;
#if (defined CLD_BASIC) && (defined BASIC_SERVER)
/* Basic Cluster - Server */
tsCLD_Basic sBasicServerCluster;
#endif
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
/* Identify Cluster - Server */
tsCLD_Identify sIdentifyServerCluster;
tsCLD_IdentifyCustomDataStructure
sIdentifyServerCustomDataStructure;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
tsCLD_ZllCommission sZllCommissionServerCluster;
tsCLD_ZllCommissionCustomDataStructure
sZllCommissionServerCustomDataStructure;
#endif
/* Mandatory client clusters */
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
/* Identify Cluster - Client */
tsCLD_Identify sIdentifyClientCluster;
```

(continues on next page)

(continued from previous page)

```
tsCLD_IdentifyCustomDataStructure
sIdentifyClientCustomDataStructure;
#endif
#if (defined CLD_BASIC) && (defined BASIC_CLIENT)
/* Basic Cluster - Client */
tsCLD_Basic sBasicClientCluster;
#endif
#if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
/* On/Off Cluster - Client */
tsCLD_OnOff sOnOffClientCluster;
#endif
#if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
/* Level Control Cluster - Client */
tsCLD_LevelControl sLevelControlClientCluster;
tsCLD_LevelControlCustomDataStructure
sLevelControlClientCustomDataStructure;
#endif
#if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_CLIENT)
/* Colour Control Cluster - Client */
tsCLD_ColourControl sColourControlClientCluster;
tsCLD_ColourControlCustomDataStructure
sColourControlClientCustomDataStructure;
#endif
/* Optional client cluster */
#if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
/* Groups Cluster - Client */
tsCLD_Groups sGroupsClientCluster;
tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
#endif
#if (defined CLD_OTA) && (defined OTA_CLIENT)
/* OTA cluster - Client */
tsCLD_AS_Ota sCLD_OTA;
tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
tsCLD_ZllCommission sZllCommissionClientCluster;
tsCLD_ZllCommissionCustomDataStructure
sZllCommissionClientCustomDataStructure;
#endif
}tsZLO_ColourControllerDevice;
```

Parent topic: Colour Controller

Registration Function The following **eZLO_RegisterColourControllerEndPoint()** function is the endpoint registration function for a color Controller device.

```
teZCL_Status eZLO_RegisterColourControllerEndPoint(
uint8 u8EndPointIdentifier,
tfpZCL_ZCLCallBackFunction cbCallBack,
tsZLO_ColourControllerDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports a color Controller device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS de-

fined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent* pCallBackEvent);
```

Also provide a pointer to a tsZLO_ColourControllerDevice structure, described in Section 3.16.2. This structure stores all variables relating to the color Controller device associated with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one color Controller device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.16.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Colour Controller

Parent topic:*Lighting and Occupancy Device Types*

Color Scene Controller The color Scene Controller device is used in nodes that support scenes and that issue color-control commands. A typical use is to adjust the intensity or color of a lighting device, or switch it on/off. For example, it can be used as a controller for a Color Dimmable Light.

- The Device ID is 0x0810.
- The header file for the device is **colour_scene_controller.h**.
- The clusters supported by the device are listed in Supported clusters.
- The device structure, tsZLO_ColourSceneControllerDevice, is listed in Device Structure.
- The endpoint registration function for the device, **eZLO_RegisterColourSceneControllerEndPoint()**, is detailed in Registration Function.

Supported clusters The clusters supported by the Colour Scene Controller device are listed in the table below.

Table 30. Clusters for Colour Scene Controller

Server (Input) side	Client (Output) side
Mandatory	
Basic	On/Off
Identify	Identify
	Level control
	Colour control
	Scenes
Optional	
Touchlink commissioning	Touchlink commissioning
	Groups
	OTA upgrade

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

Parent topic: Color Scene Controller

Device Structure The following tsZLO_ColourSceneControllerDevice structure is the shared structure for a Colour Scene Controller device:

```
typedef struct
{
tsZCL_EndPointDefinition sEndPoint;
/* Cluster instances */
    tsZLO_ColourSceneControllerDeviceClusterInstances sClusterInstance;
#if (defined CLD_BASIC) && (defined BASIC_SERVER)
/* Basic Cluster - Server */
    tsCLD_Basic sBasicServerCluster;
#endif
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
```

(continues on next page)

(continued from previous page)

```
/* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    tsCLD_IdentifyCustomDataStructure
    sIdentifyServerCustomDataStructure;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
    tsCLD_ZllCommission sZllCommissionServerCluster;
    tsCLD_ZllCommissionCustomDataStructure
    sZllCommissionServerCustomDataStructure;
#endif
/* Mandatory client clusters */
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
/* Identify Cluster - Client */
    tsCLD_Identify sIdentifyClientCluster;
    tsCLD_IdentifyCustomDataStructure
    sIdentifyClientCustomDataStructure;
#endif
#if (defined CLD_BASIC) && (defined BASIC_CLIENT)
/* Basic Cluster - Server */
    tsCLD_Basic sBasicClientCluster;
#endif
#if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
/* On/Off Cluster - Client */
    tsCLD_OnOff sOnOffClientCluster;
#endif
#if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
/* Level Control Cluster - Client */
    tsCLD_LevelControl sLevelControlClientCluster;
    tsCLD_LevelControlCustomDataStructure
    sLevelControlClientCustomDataStructure;
#endif
#if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_CLIENT)
/* Colour Control Cluster - Client */
    tsCLD_ColourControl sColourControlClientCluster;
    tsCLD_ColourControlCustomDataStructure
    sColourControlClientCustomDataStructure;
#endif
#if (defined CLD_SCENES) && (defined SCENES_CLIENT)
/* Scenes Cluster - Client */
    tsCLD_Scenes sScenesClientCluster;
    tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
#endif
/* Recommended Optional Client Cluster */
#if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
/* Groups Cluster - Client */
    tsCLD_Groups sGroupsClientCluster;
    tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
#endif
#if (defined CLD_OTA) && (defined OTA_CLIENT)
/* OTA cluster - Client */
    tsCLD_AS_Ota sCLD_OTA;
    tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
    tsCLD_ZllCommission sZllCommissionClientCluster;
    tsCLD_ZllCommissionCustomDataStructure
    sZllCommissionClientCustomDataStructure;
#endif
}tsZLO_ColourSceneControllerDevice;
```

Parent topic: Color Scene Controller

Registration Function The following **eZLO_RegisterColourSceneControllerEndPoint()** function is the endpoint registration function for a color Scene Controller device.

```
teZCL_Status eZLO_RegisterColourSceneControllerEndPoint(
uint8 u8EndPointIdentifier,
tfpZCL_ZCLCallBackFunction cbCallBack,
tsZLO_ColourSceneControllerDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports a color Scene Controller device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_ColourSceneControllerDevice structure, described in Section 3.17.2. This structure stores all variables relating to the color Controller device associated with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one color Scene Controller device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.17.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0

7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Color Scene Controller

Parent topic:*Lighting and Occupancy Device Types*

Non-Colour Controller The Non-Colour Controller device is used in nodes that issue control commands that are not related to colour - for example, to control a Dimmable Light.

- The Device ID is 0x0820.
- The header file for the device is **non_colour_controller.h**.
- The clusters supported by the device are listed in Section 3.18.1.
- The device structure, tsZLO_NonColourControllerDevice, is listed in Section 3.18.2.
- The endpoint registration function for the device, **eZLO_RegisterNonColourControllerEndPoint()**, is detailed in Section 3.18.3.

Supported clusters The clusters supported by the Non-Colour Controller device are listed in the table below.

Server (Input) side	Client (Output) side
Mandatory	
Basic	On/Off
Identify	Identify
	Level control
Optional	
Touchlink commissioning	Touchlink commissioning
	Groups
	OTA upgrade

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

Parent topic: Non-Colour Controller

Device structure The following tsZLO_NonColourControllerDevice structure is the shared structure for a Non-Colour Controller device:

```
typedef struct
{
tsZCL_EndPointDefinition sEndPoint;
/* Cluster instances */
tsZLO_NonColourControllerDeviceClusterInstances sClusterInstance;
#if (defined CLD_BASIC) && (defined BASIC_SERVER)
/* Basic Cluster - Server */
tsCLD_Basic sBasicServerCluster;
#endif
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
/* Identify Cluster - Server */
tsCLD_Identify sIdentifyServerCluster;
tsCLD_IdentifyCustomDataStructure sIdentifyServerCustomDataStructure;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
tsCLD_ZllCommission sZllCommissionServerCluster;
tsCLD_ZllCommissionCustomDataStructure
sZllCommissionServerCustomDataStructure;
#endif
/* Mandatory client clusters */
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
/* Identify Cluster - Client */
tsCLD_Identify sIdentifyClientCluster;
tsCLD_IdentifyCustomDataStructure sIdentifyClientCustomDataStructure;
#endif
#if (defined CLD_BASIC) && (defined BASIC_CLIENT)
/* Basic Cluster - Client */
tsCLD_Basic sBasicClientCluster;
#endif
#if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
/* On/Off Cluster - Client */
tsCLD_OnOff sOnOffClientCluster;
#endif
#if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
/* Level Control Cluster - Client */
tsCLD_LevelControl sLevelControlClientCluster;
tsCLD_LevelControlCustomDataStructure
sLevelControlClientCustomDataStructure;
#endif
#if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
/* Groups Cluster - Client */
tsCLD_GroupssGroup sClientCluster;
tsCLD_GroupsCustomDataStructuresGroup sClientCustomDataStructure;
#endif
#if (defined CLD_OTA) && (defined OTA_CLIENT)
/* OTA cluster - Client */
tsCLD_AS_Ota sCLD_OTA;
tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
tsCLD_ZllCommission sZllCommissionClientCluster;
tsCLD_ZllCommissionCustomDataStructure
sZllCommissionClientCustomDataStructure;
#endif
}tsZLO_NonColourControllerDevice;
```

Parent topic: Non-Colour Controller

Registration function The following **eZLO_RegisterNonColourControllerEndPoint()** function is the endpoint registration function for a Non-Colour Controller device.

```
teZCL_Status eZLO_RegisterNonColourControllerEndPoint(
            uint8 u8EndPointIdentifier,
    tfpZCL_ZCLCallBackFunction cbCallBack,
        tsZLO_NonColourControllerDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports a Non-Colour Controller device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
        (tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_NonColourControllerDevice structure, described in Section 3.18.2. This structure stores all variables relating to the color Controller device associated with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one Non-Colour Controller device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.18.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL

8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Non-Colour Controller

Parent topic:*Lighting and Occupancy Device Types*

Non-Colour Scene Controller The Non-Colour Scene Controller device is used in nodes that support 'scenes' and issues control commands which are not related to colour - for example, to control a Dimmable Light.

- The Device ID is 0x0830.
- The header file for the device is **non_colour_scene_controller.h**.
- The clusters supported by the device are listed in Supported clusters.
- The device structure, tsZLO_NonColourSceneRemoteDevice, is listed in Device Structure.
- The endpoint registration function for the device, **eZLO_RegisterNonColourSceneControllerEndPoint()**, is detailed in Registration Function.

Supported clusters The clusters supported by the Non-Colour Scene Controller device are listed in the table below.

Table: Clusters for Non-Colour Scene Controller

Server (Input) side	Client (Output) side
Mandatory	
Basic	On-Off
Identify	Identify
	Level control
	Scenes
Optional	
Touchlink commissioning	Touchlink commissioning
	Groups
	OTA upgrade

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

Parent topic: Non-Colour Scene Controller

Device Structure The following tsZLO_NonColourSceneControllerDevice structure is the shared structure for a Non-Colour Scene Controller device:

```
typedef struct
{
tsZCL_EndPointDefinition sEndPoint;
/* Cluster instances */
tsZLO_NonColourSceneControllerDeviceClusterInstances sClusterInstance;
#if (defined CLD_BASIC) && (defined BASIC_SERVER)
/* Basic Cluster - Server */
tsCLD_Basic sBasicServerCluster;
#endif
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
/* Identify Cluster - Server */
tsCLD_Identify sIdentifyServerCluster;
tsCLD_IdentifyCustomDataStructure
sIdentifyServerCustomDataStructure;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
tsCLD_ZllCommission sZllCommissionServerCluster;
tsCLD_ZllCommissionCustomDataStructure
sZllCommissionServerCustomDataStructure;
#endif
/* Mandatory client clusters */
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
/* Identify Cluster - Client */
tsCLD_Identify sIdentifyClientCluster;
tsCLD_IdentifyCustomDataStructure
sIdentifyClientCustomDataStructure;
#endif
#if (defined CLD_BASIC) && (defined BASIC_CLIENT)
/* Basic Cluster - Client */
tsCLD_Basic sBasicClientCluster;
#endif
#if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
/* On/Off Cluster - Client */
tsCLD_OnOff sOnOffClientCluster;
#endif
#if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
/* Level Control Cluster - Client */
tsCLD_LevelControl sLevelControlClientCluster;
tsCLD_LevelControlCustomDataStructure
sLevelControlClientCustomDataStructure;
#endif
#if (defined CLD_SCENES) && (defined SCENES_CLIENT)
/* Scenes Cluster - Client */
tsCLD_Scenes sScenesClientCluster;
tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
#endif
/* Recommended Optional Client Cluster*/
#if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
/* Groups Cluster - Client */
tsCLD_Groups sGroupsClientCluster;
tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
#endif
#if (defined CLD_OTA) && (defined OTA_CLIENT)
/* OTA cluster - Client */
tsCLD_AS_Ota sCLD_OTA;
tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
tsCLD_ZllCommission sZllCommissionClientCluster;
```

(continues on next page)

(continued from previous page)

```
tsCLD_ZllCommissionCustomDataStructure
sZllCommissionClientCustomDataStructure;
#endif
}tsZLO_NonColourSceneControllerDevice;
```

Parent topic: Non-Colour Scene Controller

Registration Function The following **eZLO_RegisterNonColourSceneControllerEndPoint()** function is the endpoint registration function for a Non-Colour Scene Controller device.

```
teZCL_Status eZLO_RegisterNonColourSceneControllerEndPoint(
uint8 u8EndPointIdentifier,
tfpZCL_ZCLCallBackFunction cbCallBack,
tsZLO_NonColourSceneControllerDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports a Non-Colour Scene Controller device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_NonColourSceneControllerDevice structure, described in Device Structure. This structure stores all variables relating to the color Controller device associated with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one Non-Colour Scene Controller device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Device Structure). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Non-Colour Scene Controller

Parent topic:*Lighting and Occupancy Device Types*

Control Bridge The Control Bridge device is used in nodes that relay control commands issued from another network, for example, in an Internet router with a ZigBee interface.

- The Device ID is 0x0840.
- The header file for the device is **control_bridge.h**.
- The clusters supported by the device are listed in Section 3.20.1.
- The device structure, tsZLO_ControlBridgeDevice, is listed in Section 3.20.2.
- The endpoint registration function for the device, **eZLO_RegisterControlBridgeEndPoint()**, is detailed in Section 3.20.3.

Supported clusters The clusters supported by the Control Bridge device are listed in the table below.

Table: Clusters for Control Bridge

Server (Input) side	Client (Output) side
Mandatory	
Basic	On/Off
Identify	Identify
	Groups
	Scenes
	Level control
	Colour control
Optional	
OTA upgrade	OTA upgrade
Touchlink commissioning	Touchlink commissioning
	Illuminance measurement
	Illuminance level sensing
	Occupancy sensing

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01).*

Parent topic: Control Bridge

Device Structure The following tsZLO_ControlBridgeDevice structure is the shared structure for a Control Bridge device:

```
typedef struct
{
tsZCL_EndPointDefinition sEndPoint;
/* Cluster instances */
tsZLO_ControlBridgeDeviceClusterInstances sClusterInstance;
#if (defined CLD_BASIC) && (defined BASIC_SERVER)
/* Basic Cluster - Server */
tsCLD_Basic sBasicServerCluster;
#endif
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
/* Identify Cluster - Server */
tsCLD_Identify sIdentifyServerCluster;
tsCLD_IdentifyCustomDataStructure sIdentifyServerCustomDataStructure;
#endif
/* Recommended Optional Server Cluster */
#if (defined CLD_OTA) && (defined OTA_SERVER)
/* OTA cluster */
tsCLD_AS_Ota sCLD_ServerOTA;
tsOTA_Common sCLD_OTA_ServerCustomDataStruct;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
tsCLD_ZllCommission sZllCommissionServerCluster;
tsCLD_ZllCommissionCustomDataStructure
sZllCommissionServerCustomDataStructure;
#endif
```

(continues on next page)

(continued from previous page)

```
/*
* Mandatory client clusters
*/
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
/* Identify Cluster - Client */
tsCLD_Identify sIdentifyClientCluster;
tsCLD_IdentifyCustomDataStructure sIdentifyClientCustomDataStructure;
#endif
#if (defined CLD_BASIC) && (defined BASIC_CLIENT)
/* Basic Cluster - Client */
tsCLD_Basic sBasicClientCluster;
#endif
#if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
/* On/Off Cluster - Client */
tsCLD_OnOff sOnOffClientCluster;
#endif
#if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
/* Level Control Cluster - Client */
tsCLD_LevelControl sLevelControlClientCluster;
tsCLD_LevelControlCustomDataStructure
sLevelControlClientCustomDataStructure;
#endif
#if (defined CLD_SCENES) && (defined SCENES_CLIENT)
/* Scenes Cluster - Client */
tsCLD_Scenes sScenesClientCluster;
tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
#endif
#if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
/* Groups Cluster - Client */
tsCLD_Groups sGroupsClientCluster;
tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
#endif
#if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_CLIENT)
/* Colour Control Cluster - Client */
tsCLD_ColourControl sColourControlClientCluster;
tsCLD_ColourControlCustomDataStructure
sColourControlClientCustomDataStructure;
#endif
/* Recommended Optional client clusters */
#if (defined CLD_OTA) && (defined OTA_CLIENT)
/* OTA cluster */
tsCLD_AS_Ota sCLD_OTA;
tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif
#if( defined CLD_ILLUMINANCE_MEASUREMENT) && (defined ILLUMINANCE_MEASUREMENT_
↪CLIENT)
/* Illuminance Measurement Cluster - Client */
tsCLD_IlluminanceMeasurement sIlluminanceMeasurementClientCluster;
#endif
#if (defined CLD_ILLUMINANCE_LEVEL_SENSING) && (defined ILLUMINANCE_LEVEL_SENSING_
↪CLIENT)
tsCLD_IlluminanceLevelSensing sIlluminanceLevelSensingClientCluster;
#endif
#if (defined CLD_OCCUPANCY_SENSING) && (defined OCCUPANCY_SENSING_CLIENT)
/* Occupancy Sensing Cluster - Client */
tsCLD_OccupancySensing sOccupancySensingClientCluster;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
tsCLD_ZllCommission sZllCommissionClientCluster;
tsCLD_ZllCommissionCustomDataStructure
sZllCommissionClientCustomDataStructure;
```

(continues on next page)

(continued from previous page)

```
#endif
}tsZLO_ControlBridgeDevice;
```

Parent topic: Control Bridge

Registration Function The following **eZLO_RegisterControlBridgeEndPoint()** function is the endpoint registration function for a Control Bridge device.

```
teZCL_Status eZLO_RegisterControlBridgeEndPoint(
uint8 u8EndPointIdentifier,
tfpZCL_ZCLCallBackFunction cbCallBack,
tsZLO_ControlBridgeDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports a Control Bridge device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_ControlBridgeDevice structure, described in Section 3.20.2. This structure stores all variables relating to the color Controller device associated with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one Control Bridge device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Section 3.20.2). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL

3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: Control Bridge

Parent topic:*Lighting and Occupancy Device Types*

On/Off Sensor The On/Off Sensor device is used in sensor nodes that issue control commands, for example, an infrared occupancy sensor.

- The Device ID is 0x0850.
- The header file for the device is **on_off_sensor.h**.
- The clusters supported by the device are listed in Supported clusters.
- The device structure, tsZLO_OnOffSensorDevice, is listed in Device structure.
- The endpoint registration function for the device, **eZLO_RegisterOnOffSensorEndPoint()**, is detailed in Registration function.

Supported clusters The clusters used by the On/Off Sensor device are listed in the table below.

Table 34. Clusters for On/Off Sensor

Server (Input) side	Client (Output) side
Mandatory	
Basic	On/Off
Identify	Identify
Optional	
Touchlink commissioning	Touchlink commissioning
	Level control
	Colour control
	Groups
	Scenes
	OTA upgrade

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01).*

Parent topic: On/Off Sensor

Device structure The following tsZLO_OnOffSensorDevice structure is the shared structure for a On/Off Sensor device:

```
typedef struct
{
  tsZCL_EndPointDefinition sEndPoint;
  /* Cluster instances */
  tsZLO_OnOffSensorDeviceClusterInstances sClusterInstance;
  #if (defined CLD_BASIC) && (defined BASIC_SERVER)
    /* Basic Cluster - Server */
    tsCLD_Basic sBasicServerCluster;
  #endif
  #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
    /* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    tsCLD_IdentifyCustomDataStructure
          sIdentifyServerCustomDataStructure;
    #endif
  /* Recommended Optional Server Cluster */
  #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
    tsCLD_ZllCommission sZllCommissionServerCluster;
    tsCLD_ZllCommissionCustomDataStructure
          sZllCommissionServerCustomDataStructure;
  #endif
/** Mandatory client clusters*/
  #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
    /* Identify Cluster - Client */
    tsCLD_Identify sIdentifyClientCluster;
    tsCLD_IdentifyCustomDataStructure
          sIdentifyClientCustomDataStructure;
  #endif
  #if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
```

(continues on next page)

(continued from previous page)

```
  /* On/Off Cluster - Client */
  tsCLD_OnOff sOnOffClientCluster;
  #endif
  /* Recommended Optional Client CLuster */
  #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
    /* Level Control Cluster - Client */
    tsCLD_LevelControl sLevelControlClientCluster;
    tsCLD_LevelControlCustomDataStructure
          sLevelControlClientCustomDataStructure;
  #endif
  #if (defined CLD_SCENES) && (defined SCENES_CLIENT)
  /* Scenes Cluster - Client */
  tsCLD_Scenes sScenesClientCluster;
  tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
  #endif
  #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
    /* Groups Cluster - Client */
    tsCLD_Groups sGroupsClientCluster;
    tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
  #endif
  #if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_CLIENT)
    /* Colour Control Cluster - Client */
    tsCLD_ColourControl sColourControlClientCluster;
    tsCLD_ColourControlCustomDataStructure
            sColourControlClientCustomDataStructure;
  #endif
  #if (defined CLD_OTA) && (defined OTA_CLIENT)
    /* OTA cluster - Client */
    tsCLD_AS_Ota sCLD_OTA;
    tsOTA_Common sCLD_OTA_CustomDataStruct;
  #endif
  #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
  tsCLD_ZllCommission sZllCommissionClientCluster;
  tsCLD_ZllCommissionCustomDataStructure
          sZllCommissionClientCustomDataStructure;
  #endif
} tsZLO_OnOffSensorDevice;
```

Parent topic: On/Off Sensor

Registration function The following **eZLO_RegisterOnOffSensorEndPoint()** function is the endpoint registration function for an On/Off Sensor device.

```
teZCL_Status eZLO_RegisterOnOffSensorEndPoint(
uint8 *u8EndPointIdentifier,
tfpZCL_ZCLCallBackFunction cbCallBack,
tsZLO_OnOffSensorDevice *psDeviceInfo);
```

Description This function is used to register an endpoint which supports an On/Off Sensor device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

While invoking this function, specify a user-defined callback function, which is invoked when

an event associated with the endpoint occurs. This callback function is defined according to the typedef:

```
typedef void(* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent *pCallBackEvent);
```

Also provide a pointer to a tsZLO_OnOffSensorDevice structure, described in Device structure. This structure stores all variables relating to the color Controller device associated with the endpoint. The sEndPoint and sClusterInstance fields of this structure are set by this function and must not be directly written to by the application.

The function is called multiple times if more than one endpoint is being used - for example, if more than one On/Off Sensor device is housed in the same hardware, sharing the same module.

Parameters

- *u8EndPointIdentifier*: Endpoint that is to be associated with the registered structure and callback function.
- *cbCallBack*: Pointer to the function that is used to indicate events to the application for this endpoint.
- *psDeviceInfo*: Pointer to the structure that acts as storage for all variables related to the device being registered on this endpoint (see Device structure). The sEndPoint and sClusterInstance fields are set by this register function for internal use and must not be written to by the application.

Returns

1. E_ZCL_SUCCESS
2. E_ZCL_FAIL
3. E_ZCL_ERR_PARAMETER_NULL
4. E_ZCL_ERR_PARAMETER_RANGE
5. E_ZCL_ERR_EP_RANGE
6. E_ZCL_ERR_CLUSTER_0
7. E_ZCL_ERR_CALLBACK_NULL
8. E_ZCL_ERR_CLUSTER_NULL
9. E_ZCL_ERR_SECURITY_RANGE
10. E_ZCL_ERR_CLUSTER_ID_RANGE
11. E_ZCL_ERR_MANUFACTURER_SPECIFIC
12. E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
13. E_ZCL_ERR_ATTRIBUTE_ID_ORDER
14. E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JNUG3132)*.

Parent topic: On/Off Sensor

Parent topic:*Lighting and Occupancy Device Types*

Note about the source code in the document Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2018-2025 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Revision history The table below lists the revisions to this document.

Document revision history

Document ID	Release date	Description
JNUG3131 v.2.5	15 April 2025	Removed support for K32W0 and K32W1 devices
JNUG3131 v.2.4	25 March 2025	Added support for RW612 devices
JNUG3131 v.2.3	24 January 2025	Added support for MCXW71 and MCXW72 devices
JNUG3131 v.2.2	28 February 2023	Added support for K32W1 devices
JNUG3131 v.2.1	21 June 2022	Updated to latest NXP documentation format
JNUG3131 v.2.0	18 November 2019	Addition of K32W041 and K32W061 devices
JNUG3131 v.1.0	21 June 2018	First release

JNUG3132

Preface

Introduction This manual describes the NXP implementation of the ZigBee Cluster Library (ZCL) for the ZigBee 3.0 standard. The manual describes the clusters from the ZCL that may be used in ZigBee 3.0 applications developed using an NXP ZigBee 3.0 Software Developer's Kit (SDK).

Prerequisites This manual assumes that you are already familiar with the concepts of ZigBee devices, clusters, and attributes. These are described in the ZigBee 3.0 Stack User Guide (JNUG3130), available from the NXP web site (see "*Support Resources*").

Chip Compatibility The ZCL software described in this manual can be used on the NXP K32W041, K32W061, K32W1, MCXW71, MCXW72, RW612, and JN518x family of wireless microcontrollers. The NXP hardware platforms K32W148-EVK, FRDM-MCXW71, FRDM-MCXW72, MCX-W71-EVK, MCX-W72-EVK, FRDM-RW612, and RD-RW612-BGA are supported.

Organization

- *Part I: Fundamentals* comprises four chapters:
  - *Chapter 1* introduces the ZigBee Cluster Library (ZCL)
  - *Chapter 2* describes some essential concepts for the ZCL, including read/write access to cluster attributes and the associated read/write functions
  - *Chapter 3* describes the event handling framework of the ZCL, including the supplied event handling function
  - *Chapter 4* describes the error handling provision of the ZCL, including the supplied error handling function
- *Part II: Common Resources* comprises three chapters:
  - *Chapter 5* details the general functions of the ZCL
  - *Chapter 6* details the general structures used by the ZCL
  - *Chapter 7* details the general enumerations used by the ZCL
- *Part III: General Clusters* comprises fifteen chapters:
  - *Chapter 8* details the **Basic** cluster
  - *Chapter 9* details the **Power Configuration** cluster
  - *Chapter 10* details the **Device Temperature Configuration** cluster
  - *Chapter 11* details the **Identify** cluster
  - *Chapter 12* details the **Groups** cluster
  - *Chapter 13* details the **Scenes** cluster
  - *Chapter 14* details the **On/Off** cluster
  - *Chapter 15* details the **On/Off Switch Configuration** cluster
  - *Chapter 16* details the **Level Control** cluster
  - *Chapter 17* details the **Alarms** cluster
  - *Chapter 18* details the **Time** cluster, as well as the use of ZCL time
  - Chapter 19 details the **Input and Output** clusters
  - *Chapter 20* details the **Poll Control** cluster
  - *Chapter 21* details the **Power Profile** cluster
  - *Chapter 22* details the **Diagnostics** cluster
- *Part IV: Measurement and Sensing Clusters* comprises eight chapters:
  - *Chapter 23* details the **Illuminance Measurement** cluster
  - *Chapter 24* details the **Illuminance Level Sensing** cluster
  - *Chapter 25* details the **Temperature Measurement** cluster
  - *Chapter 26* details the **Pressure Measurement** cluster
  - *Chapter 27* details the **Flow Measurement** cluster
  - *Chapter 28* details the **Relative Humidity Measurement** cluster

  - *Chapter 29* details the **Occupancy Sensing** cluster
  - *Chapter 30* details the **Electrical Measurement** cluster
- *Part V: Lighting Clusters* comprises two chapters:
  - *Chapter 31* details the **Colour Control** cluster
  - *Chapter 32* details the **Ballast Configuration** cluster
- *Part VI: HVAC Clusters* comprises three chapters:
  - *Chapter 33* details the **Thermostat** cluster
  - *Chapter 34* details the **Fan Control** cluster
  - *Chapter 35* details the **Thermostat UI Configuration** cluster
- *Part VII: Closure Clusters* comprises one chapter:
  - *Chapter 36* details the **Door Lock** cluster
- *Part VIII: Security and Safety Clusters* comprises three chapters:
  - *Chapter 37* details the **IAS Zone** cluster
  - *Chapter 38* details the **IAS ACE (Ancillary Control Equipment)** cluster
  - *Chapter 39* details the **IAS WD (Warning Device)** cluster
- *Part IX: Smart Energy Clusters* comprises three chapters:
  - *Chapter 40* details the **Price**cluster
  - *Chapter 41* details the **Demand-Response and Load Control**cluster
  - *Chapter 42* details the **Simple Metering** cluster
- *Part X: Commissioning Clusters* comprises two chapters:
  - *Chapter 43* details the **Commissioning** cluster
  - *Chapter 44* details the **Touchlink Commissioning** cluster
- *Part XI: Appliances Clusters* comprises four chapters:
  - *Chapter 45* details the **Appliance Control** cluster
  - *Chapter 46* details the **Appliance Identification** cluster
  - *Chapter 47* details the **Appliance Events and Alerts** cluster
  - *Chapter 48* details the **Appliance Statistics** cluster
- *Part XII: Over-The-Air Upgrade* comprises one chapter:
  - *Chapter 49* details the **OTA (Over-the-Air) Upgrade** cluster
- *Part XIII: Appendices* comprises the nine appendixes listed below:
  - *Appendix A: Mutex Callbacks*
  - *Appendix B: Attribute Reporting*
  - *Appendix C: Extended Attribute Discovery*
  - *Appendix D: Custom Endpoints*
  - *Appendix E: Manufacturer-specific Attributes and Commands*
  - *Appendix F: OTA Extension for Dual-processor Nodes*
  - *Appendix G: Glossary*

    These cover topics that include mutex callbacks, attribute reporting, attribute discovery, custom endpoints, manufacturer-specific attributes and commands. The storage of OTA upgrade applications in internal or external flash memory, OTA upgrade of

nodes comprising two processors, example code fragments, and a glossary of terms are also included.

Conventions

- Files, folders, functions and parameter types are represented in **bold** type.
- Function parameters are represented in *italics* type.
- Code fragments are represented in the Courier New typeface.
- This is a Tip. It indicates useful or practical information.

Note: This is a Note. It highlights important additional information.

CAUTION:

This is a Caution. It warns of situations that may result in equipment malfunction or damage.

Acronyms

S.No	Acronym	Description
1	ACE	Ancillary Control Equipment
2	API	Application Programming Interface
3	APDU	Application Protocol Data Unit
4	CIE	Control and Indicating Equipment
5	DRLC	Demand-Response and Load Control
6	HVAC	Heating, Ventilation and Air-Conditioning
7	IAS	Intruder Alarm System
8	OTA	Over The Air
9	SE	Smart Energy
10	UI	User Interface
11	UTC	Co-ordinated Universal Time
12	WD	Warning Device
13	ZCL	ZigBee Cluster Library

Related Documents Refer to the following documents for further information:

- JNUG3130 ZigBee 3.0 Stack User Guide
- JNUG3131 ZigBee Devices User Guide
- Connectivity Framework Reference Manual
- 075123 rev 7 ZigBee Cluster Library Specification [from ZigBee Alliance]
- 095264 ZigBee Over-the-air Upgrading Cluster [from ZigBee Alliance]

Support Resources To access online support resources such as SDKs, Application Notes and User Guides, visit the Wireless Connectivity page of the NXP web site:

- www.nxp.com/products/wireless-connectivity

All NXP resources referred to in this manual can be found at the above address, unless otherwise stated.

Part I: Fundamentals Part 1 comprises the following:

- *Chapter 1* introduces the ZigBee Cluster Library (ZCL)
- *Chapter 2* describes some essential concepts for the ZCL, including read/write access to cluster attributes and the associated read/write functions

- *Chapter 3* describes the event handling framework of the ZCL, including the supplied event handling function
- *Chapter 4* describes the error handling provision of the ZCL, including the supplied error handling function

ZigBee Cluster Library (ZCL) The ZigBee Cluster Library (ZCL) for ZigBee 3.0 contains standard clusters, as defined by the ZigBee Alliance, for use in ZigBee 3.0 applications over a diverse range of market sectors. Each cluster corresponds to a specific functionality, through a set of attributes and/or commands. Clusters can be selected from the ZCL to give an application the required set of capabilities.

The ZCL also provides a common means for applications to communicate. It defines a header and payload that sit inside the Protocol Data Unit (PDU) used for messages. It also defines attribute types (such as integers and strings), common commands (for example, for reading attributes), and default responses for indicating success or failure.

The NXP implementation of the ZCL, described in this manual, is supplied in the ZigBee 3.0 Software Developer's Kit (SDK) available via the NXP web site (see *Support Resources*). These SDKs provide only the clusters supported by NXP that are described in this manual. The ZCL is fully detailed in the *ZigBee Cluster Library Specification (075123)*, available from the ZigBee Alliance.

ZCL Member Clusters The clusters of the ZCL are divided into functional areas, for convenience. An application can use clusters from any number of these areas to make up its complete functionality. The clusters implemented by NXP are from the following areas (the associated clusters are listed in the referenced sub-sections):

- General - Section 1.1.1
- Measurement and Sensing - Section 1.1.2
- Lighting - Section 1.1.3
- Heating, Ventilation and Air-Conditioning (HVAC) - Section 1.1.4
- Closures - Section 1.1.5
- Security and Safety - Section 1.1.6
- Smart Energy - Section 1.1.7
- Commissioning - Section 1.1.8
- Appliances - Section 1.1.9
- Over-The-Air (OTA) Upgrade - Section 1.1.10

Note: Not all of the clusters from the above ZCL functional areas are available in the NXP software.

General The General clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in '*Part III: General Clusters*' of this manual.

Cluster	Cluster ID	Description
Basic	0x0(	The Basic cluster contains the basic properties of a ZigBee device for example, software and hardware versions. The Basic cluster allows the setting of user-defined properties such as location. This cluster is detailed in *Chapter 8*.
Power Configuration	0x0(	The Power Configuration cluster allows users to determine the power source details of a device and helps configure the under/over-voltage alarms. This cluster is detailed in *Chapter 9*.
Device Temperature Configuration	0x0(	The Device Temperature Configuration cluster allows information about the internal temperature of a device to be obtained and under/over-temperature alarms to be configured. This cluster is detailed in *Chapter 10*.
Identify	0x0(	The Identify cluster allows a ZigBee device to make itself known visually (for example, by flashing a light) to an observer, such as a network installer. This cluster is detailed in *Chapter 11*.
Groups	0x0(	The Groups cluster allows the management of the Group table concerned with group addressing - that is, the targeting of multiple endpoints using a single address. This cluster is detailed in *Chapter 12*.
Scenes	0x0(	The Scenes cluster allows the management of pre-defined sets of cluster attribute values called scenes, where a scene can be stored, retrieved, and applied to put the system into a pre-determined state. This cluster is detailed in *Chapter 13*.
On/Off	0x0(	The On/Off cluster allows a device to be put into the 'on' and 'off' states, or toggled between the two states. This cluster is detailed in *Chapter 14*.
On/Off Switch Configuration	0x0(	The On/Off Switch Configuration cluster allows the switch type on a device to be defined, as well as the commands to be generated when the switch is moved between its two states. This cluster is detailed in *Chapter 15*.
Level Control	0x0(	The Level Control cluster allows control of the level of a physical quantity (for example, heat output) on a device. This cluster is detailed in *Chapter 16*.
Alarms	0x0(	The Alarms cluster is used for sending alarm notifications and the general configuration of alarms for all other clusters on the ZigBee device (individual alarm conditions are set in the corresponding clusters). This cluster is detailed in *Chapter 17*.
Time	0x0(	The Time cluster provides an interface to a real-time clock on a ZigBee device, allowing the clock time to be read and written in order to synchronize the clock to a time standard. This is the number of seconds since 0 hrs 0 mins 0 secs on 1st January 2000 UTC (Coordinated Universal Time). This cluster includes functionality for local time-zone and daylight saving time. This cluster is detailed in *Chapter 18*.
Analogue Input (Basic)	0x0(	The Analogue Input (Basic) cluster provides an interface for accessing an analog measurement. This cluster is detailed in Section 19.1.
Analogue Output (Basic)	0x0(	The Analogue Output (Basic) cluster provides an interface for setting the value of an analog output. This cluster is detailed in Section 19.2.
Binary Input (Ba-	0x0(	The Binary Input (Basic) cluster provides an interface for accessing a binary (two-state) measurement. This cluster is detailed in Section 19.3.

Parent topic:ZCL Member Clusters

Measurement and Sensing The Measurement and Sensing clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in '*Part IV: Measurement and Sensing*Clusters' of this manual.

Cluster	Cluster ID	Description
Illuminance Measurement	0x04	The Illuminance Measurement cluster provides an interface to an illuminance measuring device, allowing the configuration of measuring and the reporting of measurements. This cluster is detailed in *Chapter 23*.
Illuminance Level Sensing	0x04	The Illuminance Level Sensing cluster provides an interface to light-level sensing functionality. This cluster is detailed in *Chapter 24*.
Temperature Measurement	0x04	The Temperature Measurement cluster provides an interface to a temperature measuring device, allowing the configuration of measuring and the reporting of measurements. This cluster is detailed in *Chapter 25*.
Pressure Measurement	0x04	The Pressure Measurement cluster provides an inter-face to a pressure measuring device, allowing the configuration of measuring and the reporting of measurements. This cluster is detailed in *Chapter 26*.
Flow Measurement	0x04	The Flow Measurement cluster provides an interface to a flow measuring device for a fluid, allowing the configuration of measuring and the reporting of measurements. This cluster is detailed in *Chapter 27*.
Relative Humidity Measurement	0x04	The Relative Humidity Measurement cluster provides an interface to a humidity measuring device, allowing the configuration of relative humidity measuring and the reporting of measurements. This cluster is detailed in *Chapter 28*.
Occupancy Sensing	0x04	The Occupancy Sensing cluster provides an interface to an occupancy sensor, allowing the configuration of sensing and the reporting of status. This cluster is detailed in *Chapter 29*.
Electrical Measurement	0x0E	The Electrical Measurement cluster provides an interface for obtaining electrical measurements from a device. This cluster is detailed in *Chapter 30*.

Parent topic:ZCL Member Clusters

Lighting The Lighting clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in '*Part V: Lighting Clusters*' of this manual.

Cluster	Cluster ID	Description
Colour Control	0x03	The Colour Control cluster can be used to adjust the colour of a light (it does not govern the overall luminance of the light, as this is controlled using the Level Control cluster). This cluster is detailed in *Chapter 31*.
Ballast Configuration	0x03	The Ballast Configuration cluster can be used to configure a lighting ballast that restricts the light levels of a connected set of lamps. This cluster is detailed in *Chapter 32*.

Parent topic:ZCL Member Clusters

Heating, Ventilation, and Air-Conditioning (HVAC) The HVAC clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in '*Part VI: HVAC Clusters*' of this manual.

Cluster	Cluster ID	Description
Thermostat	0x02	The Thermostat cluster provides a means of configuring and controlling the functionality of a thermostat. This cluster is detailed in *Chapter 33*.
Fan Control	0x02	The Fan Control cluster provides a means of con-trolling the speed or state of a fan which may be part of a heating or cooling system. The cluster is detailed in *Chapter 34*.
Thermostat User Interface Configuration	0x02	The Thermostat User Interface (UI) Configuration cluster provides a means of configuring the user inter-face (keypad and/or LCD screen) for a thermostat or a thermostat controller device. This cluster is detailed in *Chapter 35*.

Parent topic:ZCL Member Clusters

Closures The Closure clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in '*Part VII: Closure Clusters*' of this manual.

Cluster	Cluster ID	Description
Door Lock	0x0101	The Door Lock cluster provides a means of representing the state of a door lock and (optionally) the door. This cluster is detailed in *Chapter 36*.

Parent topic:ZCL Member Clusters

Security and Safety The Security and Safety clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in '*Part VIII: Security and Safety Clusters*' of this manual.

Cluster	Cluster ID	Description
IAS Zone	0x05	The IAS Zone cluster provides an interface to a zone device in an IAS (Intruder Alarm System). This cluster is detailed in *Chapter 37*.
IAS ACE (Ancillary Control Equipment)	0x05	The IAS ACE cluster provides a control interface to a CIE (Control and Indicating Equipment) device in an IAS (Intruder Alarm System). This cluster is detailed in *Chapter 38*.
IAS WD (Warning Device)	0x05	The IAS WD cluster provides an interface to a Warning Device in an IAS (Intruder Alarm System). For example, a CIE (Control and Indicating Equipment) device can use the cluster to issue alarm warning indications to a Warning Device when an alarm condition is detected. This cluster is detailed in *Chapter 39*.

Parent topic:ZCL Member Clusters

Smart Energy The Smart Energy clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in '*Part IX: Smart Energy Clusters*' of this manual.

Cluster	Cluster ID	Description
Price	0x07	The Price cluster provides the mechanism for sending and receiving pricing information within a ZigBee 3.0 network. This cluster is detailed in *Chapter 40*.
Demand-Response and Load Control	0x07	The Demand-Response and Load Control (DRLC) cluster provides an interface for controlling an attached appliance that supports load control. The cluster is able to receive load control requests and act upon them - the demand-response functionality. This cluster is detailed in *Chapter 41*.
Simple Metering	0x07	The Simple Metering cluster provides a mechanism to obtain consumption data from a metering device (elec-tric, gas, water or thermal). This cluster is detailed in *Chapter 42*.

Other Smart Energy (SE) clusters, that are not available in the ZigBee 3.0 SDK, are provided in the NXP ZigBee Smart Energy SDK. The ZigBee 3.0 SDK contains only the SE clusters that are supported by NXP for non-SE applications.

Parent topic:ZCL Member Clusters

Commissioning The Commissioning clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in '*Part X: Commissioning Clusters*' of this manual.

Cluster	Cluster ID	Description
Commissioning	0x0C	The Commissioning cluster can be used for commissioning the ZigBee stack on a device during network installation and defining the device behaviour with respect to the ZigBee network (it does not affect applications operating on the devices). This cluster is detailed in *Chapter 43*.
Touchlink Commissioning	0x1C	The Touchlink Commissioning cluster is used when forming a ZigBee 3.0 network or adding a new node to an existing network. This cluster is detailed in *Chapter 44*.

Parent topic:ZCL Member Clusters

Appliances The Appliances clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in '*Part XI: Appliances Clusters*' of this manual.

Cluster	Cluster ID	Description
Appliance Control	0x001	The Appliance Control cluster provides an interface for remotely controlling appliances in the home. This cluster is detailed in *Chapter 45*.
Appliance Identification	0x0B0	The Appliance Identification cluster provides an interface for obtaining and setting basic appliance information. This cluster is detailed in *Chapter 46*.
Appliance Events and Alerts	0x0B0	The Appliance Events and Alerts cluster provides an interface for the notification of significant events and alert situations. This cluster is detailed in *Chapter 47*.
Appliance Statistics	0x0B0	The Appliance Statistics cluster provides an interface for supplying statistical information about an appliance. This cluster is detailed in *Chapter 48*.

Parent topic:ZCL Member Clusters

Over-The-Air (OTA) Upgrade The Over-The-Air (OTA) Upgrade cluster is outlined in the table below and detailed in '*Part XII: Over-The-Air Upgrade*' of this manual.

Cluster	Cluster ID	Description
OTA Upgrad	0x00	The OTA Upgrade cluster provides the facility to upgrade (or downgrade or reinstall) application software on the nodes of a ZigBee 3.0 network by distributing the replacement software through the network (over the air) and updating the software with minimal interruption to node operation. This cluster is detailed in *Chapter 49*.

Parent topic:ZCL Member Clusters

Parent topic:*ZigBee Cluster Library (ZCL)*

General ZCL Resources In addition to clusters, the ZCL provides general (non-cluster-specific) resources. For example, common mechanisms are used to allow a cluster client to access (read and write to) the attributes on the cluster server - the NXP ZCL software includes C functions and structures for performing such accesses across all clusters.

The fundamental principles and mechanisms of the ZCL are presented in the rest of *Part I: Fundamentals*:

- *Chapter 2* describes essential ZCL principles, such as accessing attributes
- *Chapter 3* describes the ZCL handling of stack-related and timer-related events
- *Chapter 4* described the ZCL handling of errors.

The general resources provided by the ZCL software are detailed in *Part II: Common*Resources:

- *Chapter 5* details the core functions provided with the ZCL
- *Chapter 6* details the general ZCL structures
- *Chapter 7* details the general ZCL enumerations and status codes

Cluster-specific resources are detailed in the respective chapters for the clusters.

Note: It is possible to customize the clusters of the ZCL by introducing manufacturer-specific attributes and commands. The processes of adding custom attributes and commands are described in *Appendix E.*

Parent topic:*ZigBee Cluster Library (ZCL)*

ZCL Compile-time Options Before the application can be built, the ZCL compile-time options must be configured in the header file **zcl_options.h** for the application.

Enabled Clusters All required clusters must be enabled in the options header file. For example, to enable the Basic and Time clusters, the following lines are required:

```
#define CLD_BASIC
#define CLD_TIME
```

In addition, to include the software for a cluster client or server or both, it is necessary to select them in the options header file. For example, to select the Basic cluster client and server, the following lines are required:

```
#define BASIC_CLIENT
#define BASIC_SERVER
```

Support for Attribute Read/Write Read/write access to cluster attributes must be explicitly compiled into the application, and must be enabled separately for the server and client sides of a cluster using the following macros in the options header file:

```
#define ZCL_ATTRIBUTE_READ_SERVER_SUPPORTED
#define ZCL_ATTRIBUTE_READ_CLIENT_SUPPORTED
#define ZCL_ATTRIBUTE_WRITE_SERVER_SUPPORTED
#define ZCL_ATTRIBUTE_WRITE_CLIENT_SUPPORTED
```

Each of the above definitions apply to all clusters used in the application.

Tip: If only read access to attributes is required, then it is recommended not to enable write access. This is because omitting the write options gives the benefit of a reduced application size.

Optional Attributes Many clusters have optional attributes that may be enabled at compile-time via the options header file - for example, to enable the Time Zone attribute in the Time cluster:

```
#define E_CLD_TIME_ATTR_TIME_ZONE
```

Note: Cluster-specific compile-time options are described in detail in the chapters for the individual clusters. The attribute reporting feature also has its own compile-time options (see Appendix B.3.1).

Number of Endpoints The following line can be added to set the number of endpoints supported on a node (to n):

```
#define ZCL_NUMBER_OF_ENDPOINTS  n
```

The default value is 1.

Default Responses The following line can be added to enable default responses (see Section 2.5):

```
#define ZCL_DISABLE_DEFAULT_RESPONSES
```

By default, they are disabled.

APS Acknowledgements The following line can be added to enable APS acknowledgments:

```
#define ZCL_DISABLE_APS_ACK
```

By default, they are disabled.

Cooperative Tasks By default, tasks within the application are cooperative, and events will therefore not be generated for locking and unlocking mutexes for resources that are shared between the tasks. To disable the cooperative task feature, add the following line:

```
#undefine COOPERATIVE
```

Note that this requires an undefine.

Parameter Checking Parameter checking in various functions can be enabled by including the following line:

```
#define STRICT_PARAM_CHECK
```

This feature is useful for testing during application development. When the testing is complete, the option should be disabled to eliminate the checks and to save code memory. This option can be defined in the **zcl_options.h** file or the makefile.

'Wild Card' Profile Commands with a 'wild card' application profile (Profile ID of 0xFFFF) can be accepted and processed by the receiving device by including the following line:

```
#define ZCL_ALLOW_WILD_CARD_PROFILE
```

Parent topic:*ZigBee Cluster Library (ZCL)*

ZCL Fundamentals and Features This chapter describes essential ZCL concepts, including the use of shared device structures as well as remote read and write accesses to cluster attributes.

Note:

1. This chapter assumes that you are familiar with ZigBee clusters and associated concepts (such as the cluster server and client). For an introduction to ZigBee clusters, refer to the *ZigBee 3.0 Stack User Guide (JNUG3130)*.
2. The ZCL functions referred to in this chapter are detailed in *Chapter 5*.

Initializing the ZCL The ZCL can be initialized using the function **eZCL_Initialise()**, which must be called before registering any endpoints. The initialization is done using the device-specific endpoint registration functions and before starting the ZigBee PRO stack. As part of this initialization, you must specify a user-defined callback function that would be invoked when a ZigBee PRO stack event occurs that is not associated with an endpoint. Also provide a local pool of Application Protocol Data Units (APDUs) that are used by the ZCL to hold messages that are to be sent and received.

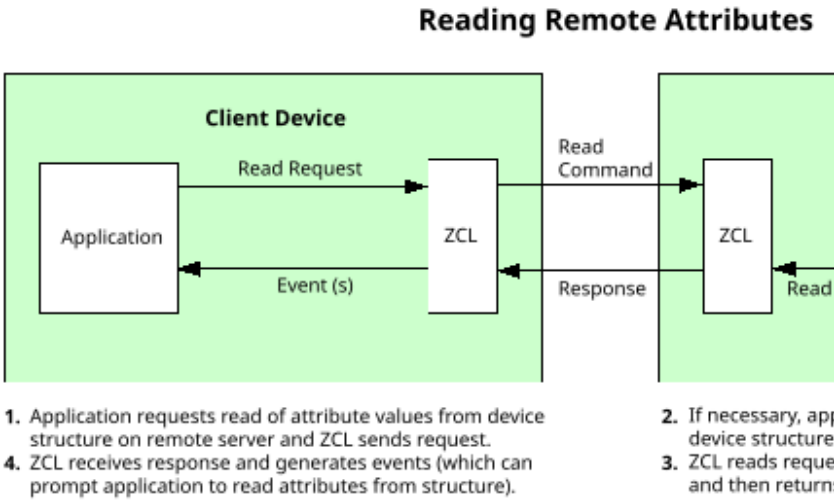
Parent topic:*ZCL Fundamentals and Features*

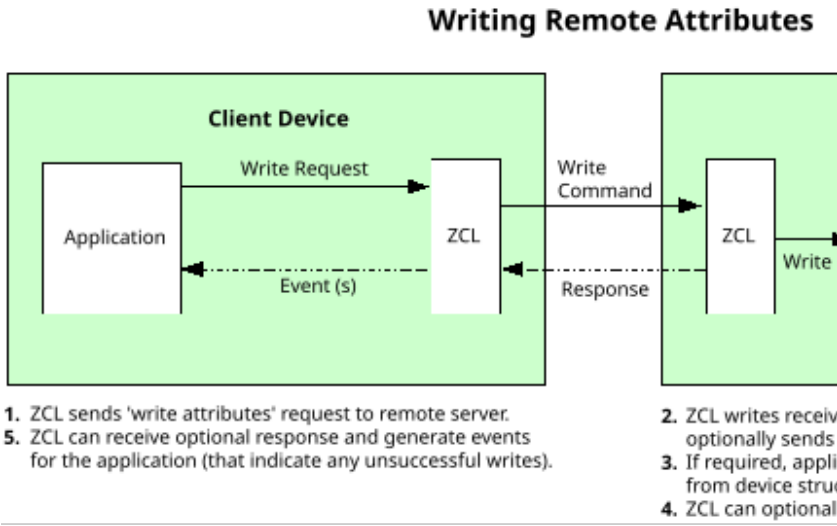
Shared Device Structures In each ZigBee device, cluster attribute values are exchanged between the application and the ZCL by means of a shared structure. This structure is protected by a mutex - see *Appendix A*. The structure for a particular ZigBee device contains structures for the clusters supported by that device.

Note: In order to use a cluster which is supported by a device, the relevant option for the cluster must be specified at build-time - see Section 1.3.

A shared device structure within a device can be accessed both by the local application and by a remote application on another device. Remote read and write operations involving a shared device structure are illustrated in the figure below. Normally, a cluster client requests these operations and they are performed on a cluster server. For more detailed descriptions of these operations, refer to Section 2.3.

Usually, the ZCL parses remote commands that write attribute values to the shared device structure. The written values can then be read by the local application. For example, an On/Off Switch device remotely writes to the shared device structure in an On/Off Light device and the local application then reads this data to change the state or configuration of the light.





Operations using Shared Device Structure

Note: Provided that there are no remote attribute writes, the attributes of a cluster server (in the shared structure) on a device are maintained by the local application(s).

Parent topic:*ZCL Fundamentals and Features*

Accessing Attributes This section describes the processes of reading and writing cluster attributes on a remote node. For the attribute access function descriptions, refer to Section 5.2.

Attribute Access Permissions For each attribute of a cluster, access permissions should be defined for the different types of access to the attribute. These permissions are configured using control flags, with one flag for each access type, as follows:

Access Type	Flag	Description
Read	E_ZCL_AF_	Global commands can read the attribute value
Write	E_ZCL_AF_	Global commands can write a new value to the attribute
Report	E_ZCL_AF_	Global commands can report the value of the attribute or configure the attribute for default reporting
Scene	E_ZCL_AF_	The attribute can be accessed through a scene (if the Scenes cluster is implemented on the same endpoint)

If a particular access type is required for an individual attribute, the corresponding flag must be defined for that attribute. This is done in the C header file for the cluster. For example, in the case of the On/Off cluster, the required flags must be defined for each attribute in the following structure in the **OnOff.c** file:

```
const tsZCL_AttributeDefinition asCLD_OnOffClusterAttributeDefinitions[] = {
#ifdef ONOFF_SERVER
  {E_CLD_ONOFF_ATTR_ID_ONOFF, (E_ZCL_AF_RD|E_ZCL_AF_SE|E_ZCL_AF_RP), E_ZCL_
↪BOOL,
  (uint32)(&((tsCLD_OnOff*)(0))->bOnOff),0}, /* Mandatory */
#ifdef CLD_ONOFF_ATTR_GLOBAL_SCENE_CONTROL
  {E_CLD_ONOFF_ATTR_ID_GLOBAL_SCENE_CONTROL, (E_ZCL_AF_RD), _ZCL_BOOL,
  (uint32)(&((tsCLD_OnOff*)(0))->bGlobalSceneControl),0}, /* Optional */
#endif
#ifdef CLD_ONOFF_ATTR_ON_TIME
  {E_CLD_ONOFF_ATTR_ID_ON_TIME, E_ZCL_AF_RD|E_ZCL_AF_WR), E_ZCL_UINT16,
  (uint32)(&((tsCLD_OnOff*)(0))->u16OnTime),0}, /* Optional */
#endif
#ifdef CLD_ONOFF_ATTR_OFF_WAIT_TIME
  {E_CLD_ONOFF_ATTR_ID_OFF_WAIT_TIME, (E_ZCL_AF_RD|E_ZCL_AF_WR), E_ZCL_
↪UINT16,
  (uint32)(&((tsCLD_OnOff*)(0))->u16OffWaitTime),0}, /* Optional */
#endif
#ifdef CLD_ONOFF_ATTR_STARTUP_ONOFF
  /* ZLO extension for OnOff Cluster */
  {E_CLD_ONOFF_ATTR_ID_STARTUP_ONOFF, E_ZCL_AF_RD|E_ZCL_AF_WR), E_ZCL_
↪ENUM8,
  (uint32)(&((tsCLD_OnOff*)(0))->eStartUpOnOff),0}, /* Optional */
#endif
#endif
  {E_CLD_GLOBAL_ATTR_ID_CLUSTER_REVISION, (E_ZCL_AF_RD|E_ZCL_AF_GA), E_ZCL_
↪UINT16,
  (uint32)(&((tsCLD_OnOff*)(0))->u16ClusterRevision),0}, /* Mandatory */
  #if (defined ONOFF_SERVER) && (defined CLD_ONOFF_ATTR_ATTRIBUTE_REPORTING_
↪STATUS)
  {E_CLD_GLOBAL_ATTR_ID_ATTRIBUTE_REPORTING_STATUS,(E_ZCL_AF_RD|E_ZCL_AF_
↪GA), E_ZCL_ENUM8,
  (uint32)(&((tsCLD_OnOff*)(0))->u8AttributeReportingStatus),0}, /* Optional */
#endif
};
```

Note: The flag *E_ZCL_AF_GA* indicates a global attribute.

Parent topic: Accessing Attributes

Reading Attributes A ZigBee 3.0 application might require to read attribute values from a remote device. Attributes are read by sending a 'read attributes' request, normally from a client cluster to a server cluster. This request can be sent using a general ZCL function (see below) or using a function which is specific to the target cluster. The cluster-specific functions for reading attributes are covered in the chapters of this manual that describe the supported clusters.

Note: Users should enable read access to cluster attributes explicitly at compile-time as described in Section 1.3.

A ZCL function is provided for reading a set of attributes of a remote cluster instance, as described in Section 2.3.2.1. A function is also provided for reading a local cluster attribute value, as described in Section 2.3.2.2.

Reading a set of attributes of a remote cluster This section describes the use of the function **eZCL_SendReadAttributesRequest()** to send a 'read attributes' request to a remote cluster in order to obtain the values of selected attributes. The resulting activities on the source and

destination nodes are outlined below and illustrated in the figure *'Write Attributes' Request and Response* below. The events generated from a 'read attributes' request are further described in *Chapter 3*.

Note: The described sequence is similar when using the cluster-specific 'read attributes' functions.

1. On Source Node The function **eZCL_SendReadAttributesRequest()** is called to submit a request to read one or more attributes on a cluster on a remote node. The information required by this function includes the following:

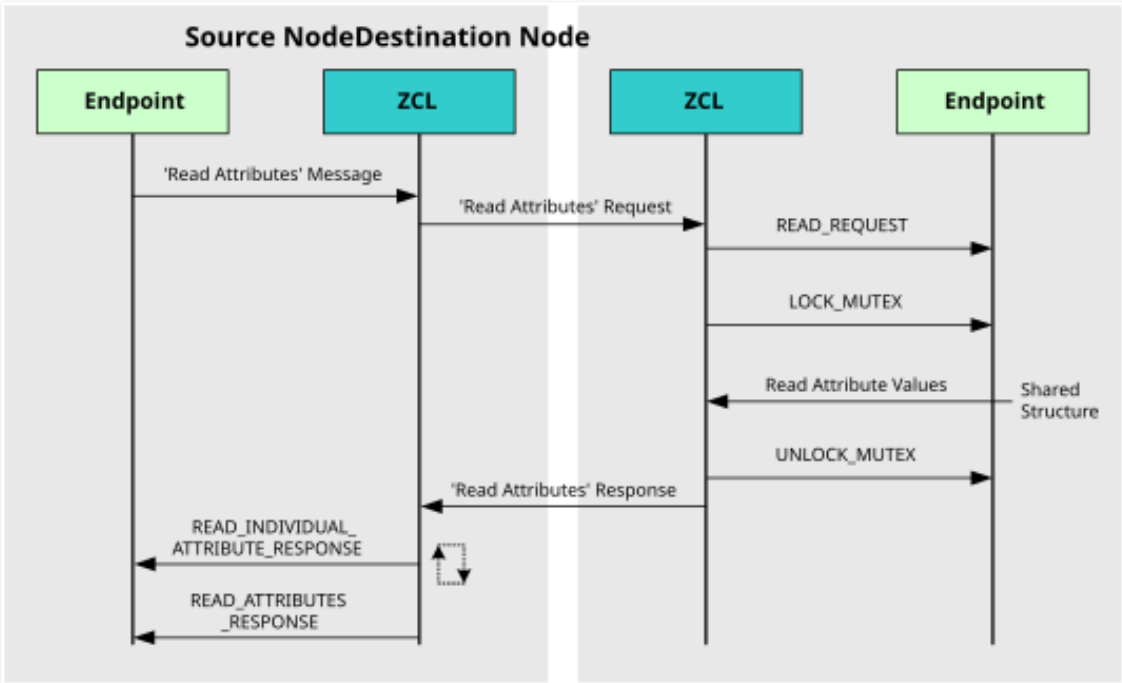
- Source endpoint (from which the read request is to be sent)
- Address of destination node for request
- Destination endpoint (on destination node)
- Identifier of the cluster containing the attributes [enumerations provided]
- Number of attributes to be read
- Array of identifiers of attributes to be read [enumerations provided]

2. On Destination Node On receiving the 'read attributes' request, the ZCL software on the destination node performs the following steps:

1. Generates an E_ZCL_CBET_READ_REQUEST event for the destination endpoint callback function which, if required, can update the shared device structure that contains the attributes to be read, before the read takes place.
2. If tasks within the application are not cooperative, the ZCL generates an E_ZCL_CBET_LOCK_MUTEX event for the endpoint callback function, which should lock the mutex that protects the shared device structure - for information on mutexes, refer to *Appendix A.*
3. Reads the relevant attribute values from the shared device structure and creates a 'read attributes' response message containing the read values.
4. If tasks within the application are not cooperative, the ZCL generates an E_ZCL_CBET_UNLOCK_MUTEXevent for the endpoint callback function, which should now unlock the mutex that protects the shared device structure (other application tasks can now access the structure).
5. Sends the 'read attributes' response to the source node of the request.

3. On Source Node On receiving the 'read attributes' response, the ZCL software on the source node performs the following steps:

1. For each attribute listed in the 'read attributes' response, it generates an E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE message for the source endpoint callback function, which may or may not take action on this message.
2. On completion of the parsing of the 'read attributes' response, it generates a single E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE message for the source endpoint callback function, which may or may not take action on this message. **'Read Attributes' Request and Response**



Note: The 'read attributes' requests and responses arrive at their destinations as data messages. Such a message triggers a stack event of the type ZPS_EVENT_APS_DATA_INDICATION, which is handled as described in Section 3.2.

Parent topic: Reading Attributes

Reading an Attribute of a Local Cluster An individual attribute of a cluster on the local node can be read using the function **eZCL_ReadLocalAttributeValue()**. The read value is returned by the function (in a memory location for which a pointer must be provided).

Parent topic: Reading Attributes

Parent topic: Accessing Attributes

Writing Attributes The ZCL provides functions for writing attribute values to both remote and local clusters, as described in Section 2.3.3.1 and Section 2.3.3.2 respectively.

Writing to Attributes of a Remote Cluster A ZigBee 3.0 application might require to write attribute values to a remote device. Attribute values are written by sending a 'write attributes' request, normally from a client cluster to a server cluster, where the relevant attributes in the shared device structure are updated. Write access to cluster attributes must be explicitly enabled at compile time as described in Section 1.3.

Three 'write attributes' functions are provided in the ZCL:

- **eZCL_SendWriteAttributesRequest():** This function sends a 'write attributes' request to a remote device, which attempts to update the attributes in its shared structure. The remote device generates a 'write attributes' response to the source device, indicating success or listing error codes for any attributes that it could not update.
- **eZCL_SendWriteAttributesNoResponseRequest():** This function sends a 'write attributes' request to a remote device, which attempts to update the attributes in its shared structure. However, the remote device does not generate a 'write attributes' response, regardless of whether there are errors.

- **eZCL_SendWriteAttributesUndividedRequest():** This function sends a 'write attributes' request to a remote device, which checks that all the attributes can be written to without error:
  - If all attributes can be written without error, all the attributes are updated.
  - If any attribute is in error, all the attributes are left at their existing values.
    - The remote device generates a 'write attributes' response to the source device, indicating success or listing error codes for attributes that are in error.

The activities surrounding a 'write attributes' request on the source and destination nodes are outlined below and illustrated in 1](reading_a_set_of_attributes_of_a_remote_cluster.md#fig2). The events generated from a 'write attributes' request are further described in *Chapter 3*.

1. On Source Node In order to send a 'write attributes' request, the application on the source node calls one of the above ZCL 'write attributes' functions to submit a request to update the relevant attributes on a cluster on a remote node. The information required by this function includes the following:

- Source endpoint (from which the write request is to be sent)
- Address of destination node for request
- Destination endpoint (on destination node)
- Identifier of the cluster containing the attributes [enumerations provided]
- Number of attributes to be written
- Array of identifiers of attributes to be written [enumerations provided]

2. On Destination Node On receiving the 'write attributes' request, the ZCL software on the destination node performs the following steps:

1. For each attribute to be written, generates an E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE event for the destination endpoint callback function.

- If required, the callback function can do either or both of the following:
  - Check that the new attribute value is in the correct range - if the value is out-of-range, the function should set the eAttributeStatus field of the event to E_ZCL_ERR_ATTRIBUTE RANGE
  - Block the write by setting the the eAttributeStatus field of the event to E_ZCL_DENY_ATTRIBUTE_ACCESS
- In the case of an out-of-range value or a blocked write, there is no further processing for that particular attribute following the 'write attributes' request.

2. If tasks within the application are not cooperative, the ZCL generates an E_ZCL_CBET_LOCK_MUTEX event for the endpoint callback function, which should lock the mutex that protects the relevant shared device structure - for information on mutexes, refer to *Appendix A.*
3. Writes the relevant attribute values to the shared device structure - an E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE event is generated for each individual attempt to write an attribute value, which the endpoint callback function can use to keep track of the successful and unsuccessful writes.

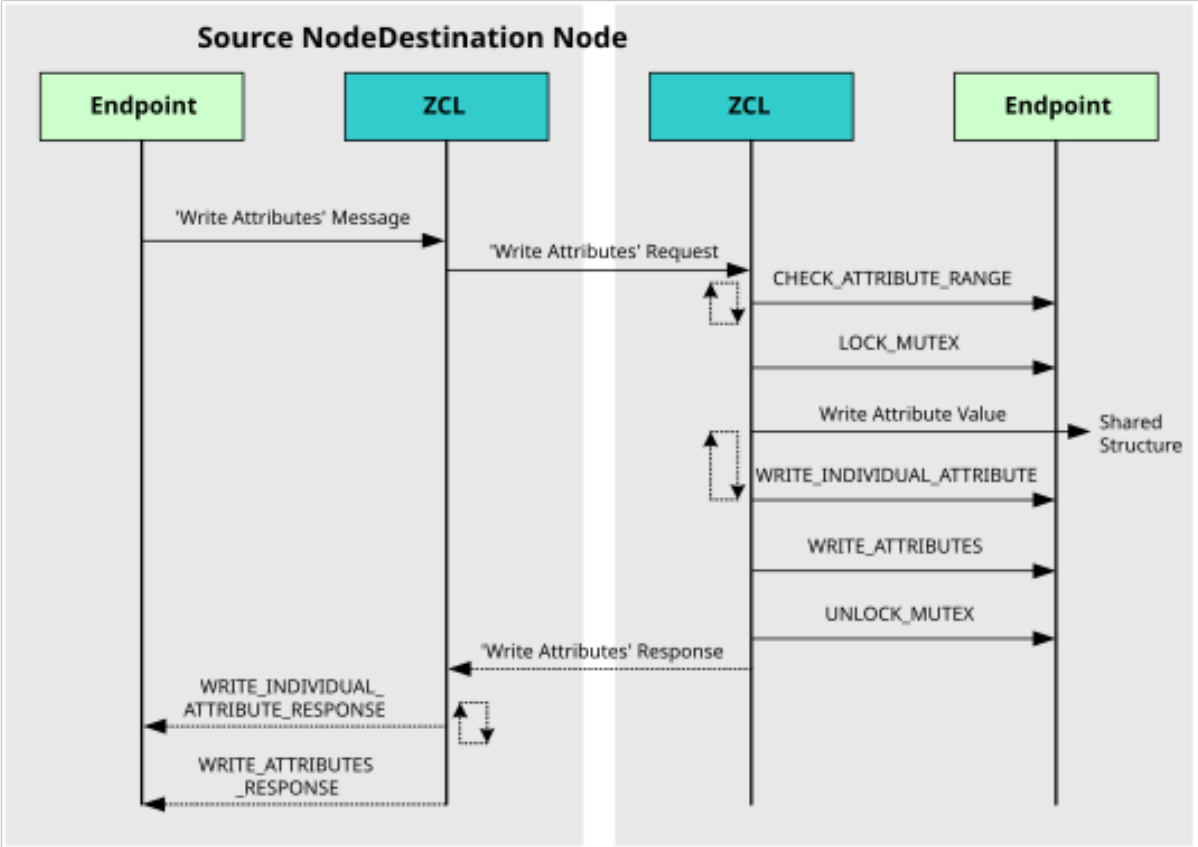
Note: If an 'undivided write attributes' request is received, an individual failed write would render the whole update process unsuccessful.

4. Generates an E_ZCL_CBET_WRITE_ATTRIBUTES event to indicate that all relevant attributes have been processed and, if required, creates a 'write attributes' response message for the source node.

5. If tasks within the application are not cooperative, the ZCL generates an E_ZCL_CBET_UNLOCK_MUTEX event for the endpoint callback function, which should now unlock the mutex that protects the shared device structure (other application tasks can now access the structure).
6. If required, sends a 'write attributes' response to the source node of the request.

3. On Source Node On receiving an optional 'write attributes' response, the ZCL software on the source node performs the following steps:

1. For each attribute listed in the 'write attributes' response, it generates an E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE message for the source endpoint callback function, which may or may not take action on this message. Only attributes for which the write has failed are included in the response and will therefore result in one of these events.
2. On completion of the parsing of the 'write attributes' response, it generates a single E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE message for the source endpoint callback function, which may or may not take action on this message.



Note: The 'write attributes' requests and responses arrive at their destinations as data messages. Such a message triggers a stack event of the type ZPS_EVENT_APS_DATA_INDICATION, which is handled as described in *Chapter 3*.

Parent topic: Writing Attributes

Writing an Attribute Value to a Local Cluster An individual attribute of a cluster on the local node can be written to using the function **eZCL_WriteLocalAttributeValue()**. The function is blocking, returning only once the value has been written.

Parent topic: Writing Attributes

Parent topic: Accessing Attributes

Attribute Discovery A ZigBee cluster may have mandatory and/or optional attributes. The desired optional attributes are enabled in the cluster structure. An application running on a cluster client may need to discover which optional attributes are supported by the cluster server.

The ZCL provides functionality to perform the necessary 'attribute discovery', as described in the rest of this section.

Note:

1. 'Extended' attribute discovery is also available. When this optional attribute is used, the accessibility of each reported attribute is also indicated. This is described in *Appendix C.*
2. Alternatively, the application on a cluster client can check whether a particular attribute exists on the cluster server by attempting to read the attribute (see Section 2.3.2) - if the attribute does not exist on the server, an error is returned.

Compile-time Options If required, the attribute discovery feature must be explicitly enabled on the cluster server and client at compile time by including the relevant defines, from those below, in the **zcl_options.h** files:

```
#define ZCL_ATTRIBUTE_DISCOVERY_SERVER_SUPPORTED
#define ZCL_ATTRIBUTE_DISCOVERY_EXTENDED_SERVER_SUPPORTED
#define ZCL_ATTRIBUTE_DISCOVERY_CLIENT_SUPPORTED
#define ZCL_ATTRIBUTE_DISCOVERY_EXTENDED_CLIENT_SUPPORTED
```

Application Coding The application on a cluster client can initiate a discovery of the attributes on the cluster server by calling the function **eZCL_SendDiscoverAttributesRequest()**, which sends a 'discover attributes' request to the server. This function allows a range of attributes to be searched for, defined by:

- The 'start' attribute in the range (the attribute identifier must be specified)
- The number of attributes in the range

Initially, the start attribute should be set to the first attribute of the cluster. If the discovery request does not return all the attributes used on the cluster server, the above function should be called again with the start attribute set to the next 'undiscovered' attribute. Multiple function calls may be required to discover all of the attributes used on the server.

On receiving a discover attributes request, the server handles the request automatically (provided that attribute discovery has been enabled in the compile-time options - see above) and replies with a 'discover attributes' response containing the requested information.

The arrival of this response at the client results in an E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_RESPON event for each attribute reported in the response. Therefore, multiple events normally result from a single discover attributes request. This event contains details of the reported attribute in a tsZCL_AttributeDiscoveryResponse structure (see Section 6.1.10).

Following the event for the final attribute reported, the event E_ZCL_CBET_DISCOVER_ATTRIBUTES_RESPONSE is generated to indicate that all attributes from the discover attributes response have been reported.

Parent topic: Accessing Attributes

Attribute Reporting A cluster client can poll the value of an attribute on the cluster server by sending a 'read attributes' request, as described in Section 2.3.2. Alternatively, the server can issue unsolicited attribute reports to the client using the 'attribute reporting' feature (in which case there is no need for the client to request attribute values).

The attribute reporting mechanism reduces network traffic compared with the polling method. It also allows a sleeping server to report its attribute values while it is awake. Attribute reporting is an optional feature and is not supported by all devices.

An 'attribute report' (from server to client) can be triggered in one of the following ways:

- by the user application (on the server device)
- automatically (triggered by a change in the attribute value or periodically)

Automatic attribute reporting for an attribute can be enabled and configured remotely from the client or, for some attributes, locally on the server (see below). If it is required, automatic attribute reporting must be enabled at compile-time on both the cluster server and client. Automatic attribute reporting is more fully described in Appendix B.1 and the configuration of attribute reporting is detailed in Appendix B.3.

The ZCL specification states that certain attributes of a cluster must be reportable. Attribute reporting for these attributes remains optional but can be enabled for the individual attributes using a flag (E_ZCL_AF_RP) in the attribute definition structure - see the example code for the On/Off cluster in Section 2.3.1. This defines those attributes that the cluster server will report by default, known as 'default reporting', but reports on other attributes can be requested/configured by the cluster client.

Note: Attribute reporting configuration data should be preserved in Non-Volatile Memory (NVM) to allow automatic attribute reporting to resume following a reset of the server device. Persisting this data in NVM is described in Appendix B.7.

An attribute report can be issued directly by the server application as follows:

- For all reportable attributes using the function **eZCL_ReportAllAttributes()**
- For an individual reportable attribute using the function **eZCL_ReportAttribute()**

Only standard attributes can be reported (this does not include manufacturer-specific attributes) and only those attributes for which reporting has been enabled. This method of attribute reporting does not require any configuration, apart from enabling reports for the desired attributes. In this case, attribute reporting does not need to be enabled at compile-time on the server, but it still needs to be enabled at compile-time on the client to allow the client to receive attribute reports.

Sending an attribute report from the server is further described in Appendix B.4 and receiving an attribute report on the client is described in Appendix B.5.

Parent topic: Accessing Attributes

Parent topic:*ZCL Fundamentals and Features*

Global Attributes There are two global attributes that are used in multiple clusters. These attributes are additions to the ZCL r6 for ZigBee 3.0 and are described below.

ClusterRevision The ClusterRevision global attribute is mandatory in all clusters. It indicates the revision of the cluster used by the current instance of the cluster on the local endpoint. The cluster specification from the ZCL r6 acts as the baseline for the revision numbering of a cluster - this is revision 1 of the cluster. The cluster revision number is incremented by one for each subsequent update of the cluster specification. Those cluster specifications that pre-date the ZCL r6 have assumed revision numbers of 0. To check that the local cluster instance is interoperable with a remote instance of the same cluster (perhaps based on a different cluster revision), the ClusterRevision attribute on the remote node should be read - if the remote cluster instance is

based on an earlier cluster revision that does not support an essential feature, the two cluster instances will not be interoperable.

AttributeReportingStatus The AttributeReportingStatus global attribute is optional and used only when attribute reporting is enabled for the cluster (see Section 2.3.5). Where 'attribute report' messages are generated for multiple attributes, this attribute indicates whether there are reports pending (0x00) or the reports are complete (0x01).

Parent topic:*ZCL Fundamentals and Features*

Default Responses The ZCL provides a default response which is generated in reply to a unicast command in the following circumstances:

- When there is no other relevant response and the requirement for default responses has not been disabled on the endpoint that sent the command.
- When an error results from a unicast command and there is no other relevant response, *even if the requirement for default responses has been disabled on the endpoint that sent the command*.

The default response disable setting is made in the bDisableDefaultResponse field of the structure tsZCL_EndPointDefinition detailed in Section 6.1.1. This setting dictates the value of the 'disable default response' bit in messages sent by the endpoint. The receiving device then uses this bit to determine whether to return a default response to the source device.

The default response includes the ID of the command that triggered the response and a status field (see Section 6.1.9). Therefore, in the case of an error, the command ID field of the default response contains the the identity of the command that caused the error.

Note: The default response can be generated on reception of all commands, including responses (for example, a 'read attributes' response) but not other default responses.

Parent topic:*ZCL Fundamentals and Features*

Handling Commands for Unsupported Clusters A node might receive a cluster-specific command or general command for a cluster that is not supported. In such a case, the ZCL sends a 'default response' containing the status code E_ZCL_CMDS_UNSUPPORTED_CLUSTER to the originator of the command. This is the standard method of handling the unsupported command (as described in the ZCL specification). Default responses are described in Section 2.5.

The NXP implementation of the ZCL provides an alternative method for dealing with commands for unsupported clusters. A user-defined callback function can be introduced which is invoked when a command is received for an unsupported cluster. This function determines whether the application will handle the command and returns a Boolean value:

- If the callback function returns TRUE, the ZCL passes the unsupported command to the application in an appropriate event. For example, consider the case when a Report Attribute command is received for the Occupancy Sensing cluster which is not supported by the device. If the callback function opts to allow the application to handle this command, the callback function returns TRUE and the ZCL then passes the command to the main application in the event *E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTE* or *E_ZCL_CBET_REPORT_ATTRIBUTES*, as appropriate.
- If the callback function returns FALSE, the ZCL handles the unsupported command in the standard way by sending a default response containing the status *E_ZCL_CMDS_UNSUPPORTED_CLUSTER*. The application is not notified about the received command.

The prototype for the user-defined callback function is as follows:

```
bool_t bZCL_OverrideHandlingEntireProfileCmd(uint16 u16ClusterId);
```

where *u16ClusterId* is the ZigBee identifier of the cluster to which the command relates.

This callback function can be registered with the ZCL using the function **vZCL_RegisterHandleGeneralCmdCallBack()**, detailed in Section 5.1.

Parent topic:*ZCL Fundamentals and Features*

Handling Commands from Other Manufacturers Every manufacturer of ZigBee Certified Products is allocated a manufacturer code by the ZigBee Alliance. The manufacturer code for NXP is 0x1037. A manufacturer-specific command that is sent by a node contains the manufacturer code for the node manufacturer.

By default, the NXP implementation of the ZCL rejects manufacturer-specific commands containing manufacturer codes other than NXP's own code. For a rejected command, the ZCL sends a 'default response' containing the status code E_ZCL_CMDS_UNSUP_MANUF_CLUSTER_COMMAND to the originator of the command. Default responses are described in Section 2.5.

However, a mechanism is available to handle multiple manufacturer codes. A user-defined callback function can be introduced, which is invoked when a manufacturer-specific command is received containing a non-NXP manufacturer code. This function determines whether the application handles the command and returns a Boolean value:

- If the callback function returns TRUE, the ZCL passes the command to the application in an appropriate event.
- If the callback function returns FALSE, the ZCL handles the command with in the standard way by sending a default response containing the status E_ZCL_CMDS_UNSUP_MANUF_CLUSTER_COMMAND. The application is not notified about the received command.

The prototype for the user-defined callback function is as follows:

```
bool_t bZCL_IsManufacturerCodeSupported(uint16 u16ManufacturerCode);
```

where *u16ManufacturerCode* is the manufacturer code in the received command.

This callback function can be registered with the ZCL using the function **vZCL_RegisterCheckForManufCodeCallBack()**, detailed in Section 5.1.

Parent topic:*ZCL Fundamentals and Features*

Bound Transmission Management ZigBee PRO provides the facility for bound transfers/transmissions. In this case, a source endpoint on one node is bound to one or more destination endpoints on other nodes. Data sent from the source endpoint is then automatically transmitted to all the bound endpoints (without the need to specify destination addresses). The bound transmission is handled by a Bind Request Server on the source node. Binding, bound transfers, and the Bind Request Server are fully described in the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

Congestion may occur if a new bound transmission is requested while the Bind Request Server is still busy completing the previous bound transmission (still sending packets to bound nodes). This causes the new bound transmission to fail. The ZCL software incorporates a feature for managing bound transmission requests, so not to overload the Bind Request Server and cause transmissions to fail.

Note: The alternative to using this feature is for the application to re-attempt bound transmissions that fail.

If this feature is enabled and a bound transmission request submitted to the Bind Request Server fails, the bound transmission APDU is automatically put into a queue. A one-second scheduler periodically takes the APDU at the head of the queue and submits it to the Bind Request Server for transmission. If this bound transmission also fails, the APDU is returned to the bound transmission queue.

The bound transmission queue has the following properties:

- Number of buffers in the queue
- Size of each buffer, in bytes

The feature is enabled and the above properties are defined at compile-time, as described below.

Note: If a single APDU does not fit into a single buffer in the queue, it is stored in multiple buffers (provided that enough buffers are available).

Compile-time Options In order to use the bound transmission management feature, the following definitions are required in the **zcl_options.h** file.

Add this line to enable the bound transmission management feature:

```
#define CLD_BIND_SERVER
```

Add this line to define the number of buffers in the bound transmission queue (in this example, the queue will contain four buffers):

```
#define MAX_NUM_BIND_QUEUE_BUFFERS  4
```

Add this line to define the size, in bytes, of a buffer in the bound transmission queue (in this example, the buffer size is 60 bytes):

```
#define MAX_PDU_BIND_QUEUE_PAYLOAD_SIZE   60
```

Certain clusters and the 'attribute reporting' feature allow APS acknowledgements to be disabled for bound transmissions. The required definitions are detailed in the cluster-specific compile-time options.

Parent topic:*ZCL Fundamentals and Features*

Command Discovery The ZCL provides the facility to discover the commands that a cluster instance on a remote device can receive and generate. This is useful since an individual cluster instance may not be able to receive or generate all of the commands that are theoretically supported by the cluster.

The commands that are supported by a cluster (and that can therefore potentially be discovered) are defined in a Command Definition table which is enabled in the cluster definition when Command Discovery is enabled (see Section 6.1.2).

Two ZCL functions are provided to implement the Command Discovery feature (as indicated in Section 2.9.1 below and fully described in Section 5.3).

Discovering Command Sets The commands supported by a remote cluster instance can be discovered as described below.

Discovering commands that can be received The commands that can be received by an instance of a cluster on a remote device can be discovered using the function

eZCL_SendDiscoverCommandReceivedRequest()

This function sends a request to the remote cluster instance, which responds with a list of commands (identified by their Command IDs). On receiving this response, the following events are generated on the local device:

- **E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_RECEIVED_RESPONSE**
  - This event is generated for each individual command reported in the response. The reported information is contained in a structure of the type tsZCL_CommandDiscoveryIndividualResponse (see Section 6.1.17).
- **E_ZCL_CBET_DISCOVER_COMMAND_RECEIVED_RESPONSE**
  - This event is generated after all the above individual events, in order to indicate the end of these events. The reported information is contained in a structure of the type tsZCL_CommandDiscoveryResponse (see Section 6.1.18).

Discovering commands that can be generated The commands that can be generated by an instance of a cluster on a remote device can be discovered using the function

eZCL_SendDiscoverCommandGeneratedRequest()

This function sends a request to the remote cluster instance, which responds with a list of commands (identified by their Command IDs). On receiving this response, the following events are generated on the local device:

- **E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_GENERATED_RESPONSE**
  - This event is generated for each individual command reported in the response. The reported information is contained in a structure of the type tsZCL_CommandDiscoveryIndividualResponse (see Section 6.1.17).
- **E_ZCL_CBET_DISCOVER_COMMAND_GENERATED_RESPONSE**
  - This event is generated after all the above individual events, in order to indicate the end of these events. The reported information is contained in a structure of the type tsZCL_CommandDiscoveryResponse (see Section 6.1.18).

Note: The above functions can be called multiple times to discover the commands in stages. After each call, the tsZCL_CommandDiscoveryResponse structure contains a Boolean flag which indicates whether there are more commands to be discovered (see Section 6.1.18). For complete details, refer to the function descriptions in Section 5.3.

Parent topic: Command Discovery

Compile-time Options If required, the Command Discovery feature must be enabled at compile-time.

To enable the feature, the following must be defined at both the local and remote ends:

```
#define ZCL_COMMAND_DISCOVERY_SUPPORTED
```

To enable the handling of Command Discovery requests and the generation of responses at the remote end, the following must be defined on the remote device:

```
#define ZCL_COMMAND_RECEIVED_DISCOVERY_SERVER_SUPPORTED
#define ZCL_COMMAND_GENERATED_DISCOVERY_SERVER_SUPPORTED
```

To enable the handling of Command Discovery responses at the local end, the following must be defined on the local device:

```
#define ZCL_COMMAND_RECEIVED_DISCOVERY_CLIENT_SUPPORTED
#define ZCL_COMMAND_GENERATED_DISCOVERY_CLIENT_SUPPORTED
```

Parent topic: Command Discovery

Parent topic:*ZCL Fundamentals and Features*

Event Handling This chapter describes the event handling framework which allows the ZCL to deal with stack-related and timer-related events (including cluster-specific events).

A message arriving in a message queue triggers a stack event whereas a timer event is triggered when a software timer expires (for more information on timer events, refer to *Section 5.2*).

The event must be wrapped in a tsZCL_CallBackEvent structure by the application (see Section 3.1 below), which then passes this event structure into the ZCL using the function **vZCL_EventHandler()**, described in Section 5.1. The ZCL processes the event and, if necessary, invokes the relevant endpoint callback function. Refer to Section 3.2 for more details of event processing.

Event Structure The tsZCL_CallBackEvent structure, in which an event is wrapped, is as follows:

```
typedef struct
{
    teZCL_CallBackEventType          eEventType;
    uint8                            u8TransactionSequenceNumber;
    uint8                            u8EndPoint;
    teZCL_Status                     eZCL_Status;
union {
    tsZCL_IndividualAttributesResponse          sIndividualAttributeResponse;
    tsZCL_DefaultResponse                       sDefaultResponse;
    tsZCL_TimerMessage                          sTimerMessage;
    tsZCL_ClusterCustomMessage                  sClusterCustomMessage;
    tsZCL_AttributeReportingConfigurationRecord
                                sAttributeReportingConfigurationRecord;
    tsZCL_AttributeReportingConfigurationResponse
                                sAttributeReportingConfigurationResponse;
    tsZCL_AttributeDiscoveryResponse          sAttributeDiscoveryResponse;
                ZCL_AttributeStatusRecord sReportingConfigurationResponse;
    tsZCL_ReportAttributeMirror sReportAttributeMirror;
    uint32              u32TimerPeriodMs;
    tsZCL_CommandDiscoveryIndividualResponse
                        sCommandsReceivedDiscoveryIndividualResponse;
    tsZCL_CommandDiscoveryResponse       sCommandsReceivedDiscoveryResponse;
    tsZCL_CommandDiscoveryIndividualResponse
                          sCommandsGeneratedDiscoveryIndividualResponse;
    tsZCL_CommandDiscoveryResponse      sCommandsGeneratedDiscoveryResponse;
    tsZCL_AttributeDiscoveryExtendedResponse
                                    sAttributeDiscoveryExtenedResponse;
}uMessage ;
    ZPS_tsAfEvent          *pZPSevent;
    tsZCL_ClusterInstance *psClusterInstance;
} tsZCL_CallBackEvent;
```

The fields of this structure are fully described Section 6.2.

In the tsZCL_CallBackEvent structure, the eEventType field defines the type of event being posted - the various event types are described in Section 3.3 below. The union and remaining fields are each relevant to only specific event types.

Parent topic:*Event Handling*

Processing Events This section outlines how the application should deal with stack events and timer events that are generated externally to the ZCL. A cluster-specific event initially arrives as one of these events.

The occurrence of an event in the ZCL queue activates a ZCL user function. The following actions must then be performed in the application:

1. The task checks whether the event that has occurred is a timer event (timer messages are collected by a user-defined function).
2. The task sets fields of the event structure tsZCL_CallBackEvent (see Section 3.1), as follows (all other fields are ignored):

- For a timer event, sets the field eEventType to E_ZCL_CBET_TIMER.
- For a millisecond timer event, sets the field eEventType to E_ZCL_CBET_TIMER_MS.
- For a stack event, sets the field eEventType to E_ZCL_ZIGBEE_EVENT and sets the field pZPSevent to point to the ZPS_tsAfEvent structure received by the application. This structure is defined in the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

3. The task passes this event structure to the ZCL using **vZCL_EventHandler()** - the ZCL then identifies the event type (see Section 3.3) and invokes the appropriate endpoint callback function.

Note: For a cluster-specific event (which arrives as a stack event or a timer event), the cluster normally contains its own event handler which is invoked by the ZCL. If the event requires the attention of the application, the ZCL replaces the eEventType field with E_ZCL_CBET_CLUSTER_CUSTOM and populates the tsZCL_ClusterCustomMessage structure with the event data. The ZCL then invokes the user-defined endpoint callback function to perform any application-specific event handling that is required.

Parent topic:*Event Handling*

Events The events that are not cluster-specific are divided into four categories (Input, Read, Write, General), as shown in the following table. The 'input events' originate externally to the ZCL and are passed into the ZCL for processing (see Section 3.2). The remaining events are generated as part of this processing.

Note: Cluster-specific events are covered in the chapter for the relevant cluster.

Category	Event
Input Events	E_ZCL_ZIGBEE_EVENT
E_ZCL_CBET_TIMER	
E_ZCL_CBET_TIMER_MS	
Read Events	E_ZCL_CBET_READ_REQUEST
E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPON	
E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE	
Write Events	E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE
E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE	
E_ZCL_CBET_WRITE_ATTRIBUTES	
E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPO	
E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE	
General Events	E_ZCL_CBET_LOCK_MUTEX
E_ZCL_CBET_UNLOCK_MUTEX	
E_ZCL_CBET_DEFAULT_RESPONSE	
E_ZCL_CBET_UNHANDLED_EVENT	
E_ZCL_CBET_ERROR	
E_ZCL_CBET_CLUSTER_UPDATE	

The above events are described below.

Input Events The 'input events' are generated externally to the ZCL. Such an event is received by the application, which wraps the event in a tsZCL_CallBackEvent structure and passes it into the ZCL using the function **vZCL_EventHandler()** - for further details of event processing, refer to Section 3.2.

- **E_ZCL_ZIGBEE_EVENT**
  - All ZigBee PRO stack events to be processed by the ZCL are designated as this type of event by setting the eEventType field in the tsZCL_CallBackEvent structure to E_ZCL_ZIGBEE_EVENT.
- **E_ZCL_CBET_TIMER**
  - A timer event (indicating that a timer has expired) which is to be processed by the ZCL is designated as this type of event by setting the eEventType field in the tsZCL_CallBackEvent structure to E_ZCL_CBET_TIMER.
- **E_ZCL_CBET_TIMER_MS**
  - A millisecond timer event (indicating that a timer has expired) which is to be processed by the ZCL is designated as this type of event by setting the eEventType field in the tsZCL_CallBackEvent structure to E_ZCL_CBET_TIMER_MS.

Read Events The 'read events' are generated as the result of a 'read attributes' request (see Section2.3.2). Some of these events are generated on the remote node and some of them are generated on the local (requesting) node, as indicated in the table below.

Generated on local node (client):	Generated on remote node (server):
	E_ZCL_CBET_READ_REQUEST
E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONS	
E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE	

The circumstances surrounding the generation of the 'read events' are outlined below:

- **E_ZCL_CBET_READ_REQUEST**
  - When a 'read attributes' request has been received and passed to the ZCL (as a stack event), the ZCL generates the event E_ZCL_CBET_READ_REQUEST for the relevant endpoint to indicate that the endpoint's shared device structure is going to be read. This gives an opportunity for the application to access the shared structure first, if required - for example, to update attribute values before they are read. This event may be ignored if the application reads the hardware asynchronously - for example, driven by a timer or interrupt.
- **E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE**
  - When a 'read attributes' response has been received by the requesting node and passed to the ZCL (as a stack event), the ZCL generates the event E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE for each individual attribute in the response. Details of the attribute are incorporated in the structure tsZCL_ReadIndividualAttributesResponse, described in Section 6.2.
  - Note that this event is often ignored by the application, while the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE (see next event) is handled.
- **E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE**
  - When a 'read attributes' response has been received by the requesting node and the ZCL has completed updating the local copy of the shared device structure, the ZCL generates the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE. The transaction sequence number and cluster instance fields of the tsZCL_CallBackEvent structure are used by this event.

Write Events The 'write events' are generated as the result of a 'write attributes' request (see Section 2.3.3). Some of these events are generated on the remote node and some of them are generated on the local (requesting) node, as indicated in the table below.

Generated on local node (client):	Generated on remote node (server):
	E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE
	E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE
	E_ZCL_CBET_WRITE_ATTRIBUTES
E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RES	
E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE	

During the process of receiving and processing a 'write attributes' request, the receiving application maintains a tsZCL_IndividualAttributesResponse structure for each individual attribute in the request:

```
typedef struct PACK {
uint16                  u16AttributeEnum;
teZCL_ZCLAttributeType  eAttributeDataType;
teZCL_CommandStatus     eAttributeStatus;
void                    *pvAttributeData;
tsZCL_AttributeStatus   *psAttributeStatus;
} tsZCL_IndividualAttributesResponse;
```

The u16AttributeEnum field identifies the attribute.

The field eAttributeDataType is set to the ZCL data type of the attribute in the request, which is checked by the ZCL to ensure that the attribute type in the request matches the expected attribute type.

The above structure is fully detailed in Section 6.2.

The circumstances surrounding the generation of the 'write events' are outlined below:

- **E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE**

  When a 'write attributes' request has been received and passed to the ZCL (as a stack event), for each attribute in the request the ZCL generates the event E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE for the relevant endpoint. This indicates that a 'write attributes' request has arrived and gives an opportunity for the application to do either or both of the following:

  - Check that the attribute value to be written falls within the valid range (range checking is not performed in the ZCL because the range may depend on application-specific rules).
  - Decide whether the requested write access to the attribute in the shared structure is allowed or not allowed.

- The value to be written is pointed to by pvAttributeData in the above structure (this does not point to the field of the shared structure containing this attribute, as the shared structure field still has its existing value).
- The attribute status field eAttributeStatus in the above structure is initially set to E_ZCL_SUCCESS. The application should set this field to E_ZCL_ERR_ATTRIBUTE_RANGE if the attribute value is out-of-range or to E_ZCL_DENY_ATTRIBUTE_ACCESS if it decides to disallow the write. Also note the following:
- If a conventional 'write attributes' request is received and an attribute value fails the range check or write access to an attribute is denied, this attribute is left unchanged in the shared structure but other attributes are updated.

- If an 'undivided write attributes' request is received and any attribute fails the range check or write access to any attribute is denied, no attribute values are updated in the shared structure.
- **E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE**
  - Following an attempt to write an attribute value to the shared structure, the ZCL generates the event E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE for the relevant endpoint. The field eAttributeStatus in the structure tsZCL_IndividualAttributesResponse indicates to the application whether the attribute value was updated successfully:
  - If the write is successful, this status field is left as E_ZCL_SUCCESS.
  - If the write is unsuccessful, this status field is set to a suitable error status (see Section 7.1.4).
- **E_ZCL_CBET_WRITE_ATTRIBUTES**
  - Once all the attributes in a 'write attributes' request have been processed, the ZCL generates the event E_ZCL_CBET_WRITE_ATTRIBUTES for the relevant endpoint.
- **E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE**
  - The E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE event is generated for each attribute that is listed in an incoming 'write attributes' response message. Only attributes that have failed to be written are contained in the message. The field eAttributeStatus of the structure tsZCL_IndividualAttributesResponse indicates the reason for the failure (see Section 7.1.4).
- **E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE**
  - The E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE event is generated when the parsing of an incoming 'write attributes' response message is complete. This event is particularly useful following a write where all the attributes have been written without errors since, in this case, no E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE event is generated.

General Events

- **E_ZCL_CBET_LOCK_MUTEX and E_ZCL_CBET_UNLOCK_MUTEX**
  - When an application task accesses the shared device structure of an endpoint, a mutex should be used by the task to protect the shared structure from conflicting accesses. Thus, the ZCL may need to lock or unlock a mutex in handling an event - for example, when a "read attributes" request has been received and passed to the ZCL (as a stack event). In these circumstances, the ZCL generates the following events:
  - E_ZCL_CBET_LOCK_MUTEX when a mutex is to be locked
  - E_ZCL_CBET_UNLOCK_MUTEX when a mutex is to be unlocked
  - The ZCL specifies one of the above events in invoking the callback function for the endpoint. Thus, the endpoint callback function must include the necessary code to lock and unlock a mutex - for further information, refer to *Appendix A.*
  - The locking and unlocking of a mutex are useful if the tasks in the application are non-cooperative while sharing the same resource. To optimize the code, the above events are not generated when the tasks are in a cooperative group. Tasks are cooperative by default and, if not required, this feature can be disabled in the **zcl_options.h** file (see Section 1.3).
- **E_ZCL_CBET_DEFAULT_RESPONSE**
  - The E_ZCL_CBET_DEFAULT_RESPONSE event is generated when a ZCL default response message has been received. These messages indicate that either an error has

occurred or a message has been processed. The payload of the default response message is contained in the structure tsZCL_DefaultResponseMessage below:

```
typedef struct PACK {
    uint8        u8CommandId;
    uint8        u8StatusCode;
} tsZCL_DefaultResponseMessage;
```

- u8CommandId is the ZCL command identifier of the command which triggered the default response message.
- u8StatusCode is the status code from the default response message. It is set to 0x00 for OK or to an error code defined in the ZCL Specification.
- **E_ZCL_CBET_UNHANDLED_EVENT and E_ZCL_CBET_ERROR**
- The E_ZCL_CBET_UNHANDLED_EVENT and E_ZCL_CBET_ERROR events indicate that a stack message has been received which cannot be handled by the ZCL. The *pZPSevent field of the tsZCL_CallBackEvent structure points to the stack event that caused the event.
- **E_ZCL_CBET_CLUSTER_UPDATE**
- The E_ZCL_CBET_CLUSTER_UPDATE event indicates that one or more attribute values for a cluster on the local device may have changed.

Note: ZCL error events and default responses (see Section 6.1.9) may be generated when problems occur in receiving commands. The possible ZCL status codes contained in the events and responses are detailed in Section 4.2.

Parent topic:*Event Handling*

Error Handling This chapter describes the error handling provision in the NXP implementation of the ZCL.

Last Stack Error The last error generated by the ZigBee PRO stack can be obtained using the ZCL function **eZCL_GetLastZpsError()**, described in Section 5.1. The possible returned errors are listed in the Return/Status Codes chapter of the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

Parent topic:*Error Handling*

Error/Command Status on Receiving Command When a device receives a command, an error might be generated. If receiving a command results in an error, an event of the type E_ZCL_CBET_ERROR is generated on the device. In such cases, the following status codes may be used:

- The ZCL status of the event (sZCL_CallBackEvent.eZCL_Status) is set to one of the error codes detailed in Section 7.2.
- A 'default response' (see Section 6.1.9) may be generated which contains one of the command status codes detailed in Section 7.1.4. This response is sent to the source node of the received command (and can be intercepted using an over-air sniffer).

The table below details the error and command status codes that may be generated.

Error Status (in Event)	Command Status (in Response)	Notes
E_ZCL_ERR_ZRECEIVE_ *	None	A receive error has occurred. This error is often security-based due to key establishment not being successfully completed - ZPS error is ZPS_APL_APS_E_SECURITY_FAIL.
E_ZCL_ERR_EP_UNKNO	E_ZCL_CMDS_S	Destination endpoint for the command is not registered with the ZCL.
E_ZCL_ERR_CLUSTER_N	E_ZCL_CMDS_U	Destination cluster for the command is not registered with the ZCL.
E_ZCL_ERR_SECURITY_	E_ZCL_CMDS_F	Attempt made to access a cluster using a packet without the necessary application-level (APS) encryption.
None	E_ZCL_CMDS_U	Command has no handler enabled in **zcl_options.h** file.
E_ZCL_ERR_CUSTOM_C	E_ZCL_CMDS_U	Custom command has no registered handler or its handler has not returned E_ZCL_SUCCESS.
E_ZCL_ERR_KEY_ESTAB MENT_END_POINT_NO	None	Key Establishment cluster has not been registered correctly.
E_ZCL_ERR_KEY_ESTAB MENT_CALLBACK_ERR	None	Key Establishment cluster callback function has returned an error.
None	E_ZCL_CMDS_M	A received message is incomplete due to some missing command-specific data.

* ZigBee PRO stack raises an error which can be retrieved using **eZCL_GetLastZpsError()**.

Parent topic:*Error Handling*

Part II: Common Resources This part comprises three chapters:

- *Chapter 5* details the general functions of the ZCL.
- *Chapter 6* details the general structures used by the ZCL.
- *Chapter 7* details the general enumerations used by the ZCL.

ZCL Functions This chapter details the core functions of the ZCL that may be needed irrespective of the clusters used. These functions include:

- General functions - see Section 5.1
- Attribute Access functions - see Section 5.2
- Command Discovery functions - see Section 5.3

General Functions This section details a set of general ZCL functions that deal with ZCL initialization, endpoint registration, timing, APS acknowledgments, event handling, and error handling:

1. eZCL_Initialise
2. eZCL_Register
3. vZCL_EventHandler
4. eZCL_Update100mS
5. vZCL_DisableAPSACK
6. eZCL_GetLastZpsError

7. vZCL_RegisterHandleGeneralCmdCallBack
8. vZCL_RegisterCheckForManufCodeCallBack

eZCL_Initialise

```
teZCL_Status eZCL_Initialise(
    tfpZCL_ZCLCallBackFunction cbCallBack,
    PDUM_thAPdu hAPdu);
```

Description This function initializes the ZCL. It should be called before registering any endpoints (using one of the device-specific endpoint registration functions) and before starting the ZigBee PRO stack.

As part of this function call, you must specify a user-defined callback function that is invoked when a ZigBee PRO stack event occurs that is not associated with an endpoint (the callback function for events associated with an endpoint is specified when the endpoint is registered using one of the registration functions). This callback function is defined according to the typedef:

```
typedef void (* tfpZCL_ZCLCallBackFunction)
        (tsZCL_CallBackEvent *pCallBackEvent);
```

You must also provide a pointer to a local pool of Application Protocol Data Units (APDUs) that would be used by the ZCL to hold messages to be sent and received.

Parameters

- *cbCallBack*: Pointer to a callback function to handle stack events that are not associated with a registered endpoint
- *hAPdu*: Pointer to a pool of APDUs for holding messages to be sent and received

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_HEAP_FAIL
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: General Functions

eZCL_Register

```
teZCL_Status eZCL_Register(
    tsZCL_EndPointDefinition *psEndPointDefinition);
```

Description This function is used to register an endpoint with the ZCL. The function validates the clusters and corresponding attributes supported by the endpoint, and registers the endpoint.

The function should only be called to register a custom endpoint (which does not contain one of the standard ZigBee device types). It should be called for each custom endpoint on the local node. The function is not required when using a standard ZigBee device (for example, On/Off Switch) on an endpoint - in this case, the appropriate device registration function should be used.

Parameters

- *psEndPointDefinition*: Pointer to tsZCL_EndPointDefinition structure for the endpoint to be registered (see Section 'tsZCL_EndPointDefinition'

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_PARAMETER_RANGE
- E_ZCL_ERR_HEAP_FAIL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_SECURITY_RANGE
- E_ZCL_ERR_CLUSTER_0
- E_ZCL_ERR_CLUSTER_NULL
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_CLUSTER_ID_RANGE
- E_ZCL_ERR_ATTRIBUTES_NULL
- E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED,
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND,
- E_ZCL_ERR_CALLBACK_NULL

Parent topic: General Functions

vZCL_EventHandler

```
void vZCL_EventHandler(
    tsZCL_CallBackEvent *psZCLCallBackEvent);
```

Description This function should be called when an event (ZigBee stack, peripheral, timer, or cluster event) occurs. The function is used to pass the event to the ZCL. The ZCL then processes the event, including a call to any necessary callback function.

The event is passed into the function in a tsZCL_CallBackEvent structure, which the application must fill in - refer to Section 6.2 for details of this structure.

Parameters

- *psZCLCallBackEvent*: Pointer to a tsZCL_CallBackEvent event structure (see Section 6.2) containing the event to process

Returns

- None

Parent topic: General Functions

eZCL_Update100mS

```
teZCL_Status eZCL_Update100mS(void);
```

Description This function is used to service all the timing needs of the clusters used by the application and should be called every 100 ms. This can be achieved by using a 100 ms software timer to periodically prompt execution of this function.

The function calls the external user-defined function **vIdEffectTick()**, which can be used to implement an identify effect on the node. This function must be defined in the application, irrespective of whether identify effects are needed (and therefore, may be empty). The function prototype is:

```
void vIdEffectTick(void)
```

Parameters

- None

Returns

- E_ZCL_SUCCESS

Parent topic: General Functions

vZCL_DisableAPSACK

```
void vZCL_DisableAPSACK(bool_t bDisableAPSACK);
```

Description This function can be used to enable/disable the request of an APS acknowledgment when a ZCL command is sent. APS acknowledgments are enabled by default.

Parameters

- *bDisableAPSACK*: Enable or disable APS acknowledgment requests:
  - TRUE - Disable requests of APS acknowledgments
  - FALSE - Enable requests of APS acknowledgments

Returns

- None

Parent topic: General Functions

eZCL_GetLastZpsError

```
ZPS_teStatus eZCL_GetLastZpsError(void);
```

Description This function returns the last error code generated by the ZigBee PRO stack when accessed from the ZCL.

For example, if a call to the On/Off cluster function **eCLD_OnOffCommandSend()** returns E_ZCL_ERR_ZTRANSMIT_FAIL (because the ZigBee PRO API function that was used to transmit the command failed), the **eZCL_GetLastZpsError()** function can be called to obtain the return code from the ZigBee PRO stack.

Note that the error code is not updated on a successful call to the ZigBee PRO stack. Also, there is only a single instance of the error code, so subsequent errors would over-write the current value.

Note: If an error occurs when a command is received, an event of type E_ZCL_CBET_ERROR is generated on the receiving node. A 'default response' may also be returned to the source node of the received command. The possible ZCL status codes in the error event and in the default response are detailed in Section 4.2.

Parameters

- None

Returns The error code of the last ZigBee PRO stack error - see the *Return/Status Codes* chapter of the *ZigBee 3.0 Stack User Guide (JNUG3130)*

Parent topic: General Functions

vZCL_RegisterHandleGeneralCmdCallBack

```
void vZCL_RegisterHandleGeneralCmdCallBack(void *fnPtr);
```

Description This function is used to register an optional user-defined callback function that is invoked when a cluster-specific or general command is received for a cluster that is not supported on the local device.

The purpose of the registered callback function is to determine whether the application would handle the unsupported command. The prototype of the callback function is as follows:

```
bool_t bZCL_OverrideHandlingEntireProfileCmd(uint16 u16ClusterId);
```

where *u16ClusterId* is the ZigBee identifier of the cluster to which the command relates. The function returns a Boolean value, which is TRUE if the main application handles the command and FALSE if the ZCL would handle the command:

- If the function returns TRUE, the ZCL passes the command to the main application in an appropriate event.
- If the function returns FALSE, the ZCL sends a 'default response' containing the status E_ZCL_CMDS_UNSUPPORTED_CLUSTER to the originator of the command (this is also the standard way of handling a command for an unsupported cluster when a callback function has not been registered).

For more information on handling commands for unsupported clusters, refer to Section 2.6.

Parameters

- *fnPtr*: Pointer to user-defined callback function to be registered

Returns

- None

Parent topic: General Functions

vZCL_RegisterCheckForManufCodeCallBack

```
void vZCL_RegisterCheckForManufCodeCallBack(void *fnPtr);
```

Description This function is used to register an optional user-defined callback function that is invoked when a manufacturer-specific command is received containing a manufacturer code other than NXP's own code (0x1037).

The purpose of the registered callback function is to determine whether the application would handle a received command with a given manufacturer code. The prototype of the callback function is as follows:

```
bool_t bZCL_IsManufacturerCodeSupported(uint16 u16ManufacturerCode);
```

where *u16ManufacturerCode* is the manufacturer code contained in the received command. The function returns a Boolean value, which is TRUE if the main application handles the command and FALSE if the ZCL handles the command:

- If the function returns TRUE, the ZCL passes the command to the main application in an appropriate event.
- If the function returns FALSE, the ZCL sends a 'default response' containing the status E_ZCL_CMDS_UNSUP_MANUF_CLUSTER_COMMAND to the originator of the command (this is also the standard way of handling a command with a non-NXP manufacturer code when a callback function has not been registered).

For more information on handling manufacturer-specific commands containing non-NXP manufacturer codes, refer to Section 2.7.

Parameters

- *fnPtr*: Pointer to user-defined callback function to be registered

Returns

- None

Parent topic: General Functions

Parent topic:*ZCL Functions*

Attribute Access Functions The following functions are provided in the ZCL for accessing cluster attributes on a remote device:

1. eZCL_SendReadAttributesRequest
2. eZCL_SendWriteAttributesRequest
3. eZCL_SendWriteAttributesNoResponseRequest
4. eZCL_SendWriteAttributesUndividedRequest
5. eZCL_SendDiscoverAttributesRequest
6. eZCL_SendDiscoverAttributesExtendedRequest

7. eZCL_SendConfigureReportingCommand
8. eZCL_SendReadReportingConfigurationCommand
9. eZCL_ReportAllAttributes
10. eZCL_ReportAttribute
11. eZCL_CreateLocalReport
12. eZCL_SetReportableFlag
13. vZCL_SetDefaultReporting
14. eZCL_HandleReadAttributesResponse
15. eZCL_ReadLocalAttributeValue
16. eZCL_WriteLocalAttributeValue
17. eZCL_OverrideClusterControlFlags
18. eZCL_SetSupportedSecurity

Note: In addition to the general function **eZCL_SendReadAttributesRequest()**, there are cluster-specific 'read attributes' functions for some clusters.

eZCL_SendReadAttributesRequest

```
teZCL_Status eZCL_SendReadAttributesRequest(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    uint16 u16ClusterId,
    bool_t bDirectionIsServerToClient,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint8 u8NumberOfAttributesInRequest,
    bool_t bIsManufacturerSpecific,
    uint16 u16ManufacturerCode,
    uint16 *pu16AttributeRequestList);
```

Description This function can be used to send a 'read attributes' request to a cluster on a remote endpoint. Read access to cluster attributes on the remote node must be enabled at compile-time as described in Section 1.3.

Specify the endpoint on the local node from which the request is to be sent. Also specify the address of the destination node, the destination endpoint number, and the cluster from which attributes are to be read. It is possible to use this function to send a request to bound endpoints or to a group of endpoints on remote nodes. In the latter case, a group address must be specified.

Note: When sending requests to multiple endpoints through a single call to this function, multiple responses would subsequently be received from the remote endpoints.

The function allows you to read selected attributes from the remote cluster. Specify the number of attributes to be read and to identify the required attributes by means of an array of identifiers. This array must be created by the application (the memory space for the array only needs to persist for the duration of this function call). The attributes can be from the relevant ZigBee cluster specification or manufacturer-specific.

Also provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This feature is useful while sending more than one request to the same destination endpoint.

On receiving the 'read attributes' response, the obtained attribute values are automatically written to the local copy of the shared device structure for the remote device and an E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE event is then generated for each attribute updated. The response may not contain values for all requested attributes. Finally, once all received attribute values have been parsed, the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE is generated.

Parameters

- *u8SourceEndPointId*Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*Number of the remote endpoint to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *u16ClusterId*Identifier of the cluster to be read (see the macros section in the cluster header file)
- *bDirectionIsServerToClient*Direction of request:
  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8NumberOfAttributesInRequest*: Number of attributes to be read
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or defined in the relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)
- *pu16AttributeRequestList*: Pointer to an array which lists the attributes to be read. The attributes are identified by using enumerations (listed in the 'Enumerations' section of each cluster-specific chapter)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN

Parent topic: Attribute Access Functions

eZCL_SendWriteAttributesRequest

```
teZCL_Status eZCL_SendWriteAttributesRequest(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    uint16 u16ClusterId,
    bool_t bDirectionIsServerToClient,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint8 u8NumberOfAttributesInRequest,
    bool_t bIsManufacturerSpecific,
    uint16 u16ManufacturerCode,
    tsZCL_WriteAttributeRecord *pu16AttributeRequestList);
```

Description This function can be used to send a 'write attributes' request to a cluster on a remote endpoint. The function also demands a 'write attributes' response from the remote endpoint, listing any attributes that could not be updated (see below). Note that write access to cluster attributes on the remote node must be enabled at compile-time as described in Section 1.3.

You must specify the endpoint on the local node from which the request is to be sent.

You must also specify the address of the destination node, the destination endpoint number and the cluster to which attributes are to be written. It is possible to use this function to send a request to bound endpoints or to a group of endpoints on remote nodes - in the latter case, a group address must be specified. Note that when sending requests to multiple endpoints through a single call to this function, multiple responses will subsequently be received from the remote endpoints.

The function allows you to write selected attributes to the remote cluster. You are required to specify the number of attributes to be written and to identify the required attributes by means of an array of identifiers - this array must be created by the application (the memory space for the array only needs to be valid for the duration of this function call). The attributes can be from the relevant ZigBee cluster specification or manufacturer-specific

You are also required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Following a 'write attributes' response from the remote endpoint, the event E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE is generated for each attribute that was not successfully updated on the remote endpoint. Finally, the event E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE is generated when processing of the response is complete. If required, these events can be handled in the user-defined callback function which is specified when the (requesting) endpoint is registered using the endpoint registration function for the device type.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent. This parameter is ignored while sending to address types, eZCL_AMBOUND and eZCL_AMGROUP.
- *u16ClusterId*: Identifier of the cluster to be written to (see the macros section in the cluster header file)
- *bDirectionIsServerToClient*: Direction of request:

  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8NumberOfAttributesInRequest*: Number of attributes to be written
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)**
- *pu16AttributeRequestList*: Pointer to an array of structures containing the attribute data to be written (see Section 6.1.21)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN

Parent topic: Attribute Access Functions

eZCL_SendWriteAttributesNoResponseRequest

```
teZCL_Status eZCL_SendWriteAttributesNoResponseRequest(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    uint16 u16ClusterId,
    bool_t bDirectionIsServerToClient,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint8 u8NumberOfAttributesInRequest,
    bool_t bIsManufacturerSpecific,
    uint16 u16ManufacturerCode,
    tsZCL_WriteAttributeRecord *pu16AttributeRequestList);
```

Description This function can be used to send a 'write attributes' request to a cluster on a remote endpoint without requiring a response. If you need a response to your request, use the function **eZCL_SendWriteAttributesRequest()** instead. Note that write access to cluster attributes on the remote node must be enabled at compile-time as described in Section 1.3.

Specify the endpoint on the local node from which the request is to be sent. Also specify the address of the destination node, the destination endpoint number, and the cluster to which attributes are to be written. It is possible to use this function to send a request to bound endpoints or to a group of endpoints on remote nodes - in the latter case, a group address must be specified.

The function allows you to write selected attributes to the remote cluster. Users should specify the number of attributes to be written and identify the required attributes by using an array of identifiers. The application should create this array. The memory space for the array only needs to be valid for the duration of this function call. The attributes can be from the relevant ZigBee cluster specification or manufacturer-specific.

You must also provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent.
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *u16ClusterId*: Identifier of the cluster to be written to (see the macros section in the cluster header file):
- *bDirectionIsServerToClient*: Direction of request:
  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8NumberOfAttributesInRequest*: Number of attributes to be written
- *bIsManufacturerSpecific*: : Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)**
- *pu16AttributeRequestList*: Pointer to an array of structures containing the attribute data to be written (see Section 6.1.21).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_FAIL

- E_ZCL_ERR_EP_UNKNOWN

Parent topic: Attribute Access Functions

eZCL_SendWriteAttributesUndividedRequest

```
teZCL_Status eZCL_SendWriteAttributesUndividedRequest(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    uint16 u16ClusterId,
    bool_t bDirectionIsServerToClient,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint8 u8NumberOfAttributesInRequest,
    bool_t bIsManufacturerSpecific,
    uint16 u16ManufacturerCode,
    tsZCL_WriteAttributeRecord *pu16AttributeRequestList);**
```

Description This function can be used to send an 'undivided write attributes' request to a cluster on a remote endpoint. This ensures that all the specified attributes are updated on the remote endpoint or none at all. This implies that if one of the attributes cannot be written, then none of them are updated. The function also demands a 'write attributes' response from the remote endpoint, indicating success or failure.

Note: Write access to cluster attributes on the remote node must be enabled at compile-time as described in Section 1.3.

You should specify the endpoint on the local node from which the request is to be sent.

You must also specify the address of the destination node, the destination endpoint number and the cluster to which attributes are to be written. It is possible to use this function to send a request to bound endpoints or to a group of endpoints on remote nodes - in the latter case, a group address must be specified. When sending requests to multiple endpoints through a single call to this function, multiple responses are subsequently received from the remote endpoints.

The function allows you to write selected attributes to the remote cluster. You must specify the number of attributes to be written and to identify the required attributes by means of an array of identifiers. The application should create this array, such that the memory space for the array only needs to be valid for the duration of this function call. The attributes can be from the relevant ZigBee cluster specification or manufacturer-specific

You must also provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Following a 'write attributes' response from the remote endpoint, the event E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE is generated to indicate success or failure. This event can be handled in the user-defined callback function which is specified when the (requesting) endpoint is registered using the appropriate endpoint registration function for the device type.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent. Note that this parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

- *u16ClusterId*: Identifier of the cluster to be written to (see the macros section in the cluster header file)
- *bDirectionIsServerToClient*: Direction of request:
  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8NumberOfAttributesInRequest*: Number of attributes to be written
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)
- *pu16AttributeRequestList*: Pointer to an array of structures containing the attribute data to be written (see Section 6.1.21)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN

Parent topic: Attribute Access Functions

eZCL_SendDiscoverAttributesRequest

```
teZCL_Status eZCL_SendDiscoverAttributesRequest(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    uint16 u16ClusterId,
    bool_t bDirectionIsServerToClient,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint16 u16AttributeId,
    bool_t bIsManufacturerSpecific,
    uint16 u16ManufacturerCode,
    uint8 u8MaximumNumberOfIdentifiers);
```

Description This function can be used to send a 'discover attributes' request to a cluster (normally a cluster server) on a remote device. The range of attributes of interest (within the standard set of cluster attributes) must be defined by specifying the identifier of the 'start' attribute and the number of attributes in the range. The function returns immediately and the results of the request are later received in a 'discover attributes' response.

You must provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful while sending more than one request to the same destination endpoint.

On receiving the 'discover attributes' response, the event

E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_RESPONSE

is generated for each attribute reported in the response. Therefore, multiple events normally result from a single function call ('discover attributes' request). Following the event for the final attribute reported, the event

E_ZCL_CBET_DISCOVER_ATTRIBUTES_RESPONSE

is generated to indicate that all attributes from the discover attributes response have been reported.

Attribute discovery is fully described in Section 2.3.4.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent
- *u16ClusterId*: Identifier of the cluster to be queried (see the macros section in the cluster header file): :
- *bDirectionIsServerToClient*: Direction of request:
  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u16AttributeId*: Identifier of 'start' attribute of interest
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)
- *u8MaximumNumberOfIdentifiers*: Number of attributes in attribute range of interest (maximum number of attributes to report in response)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL

Parent topic: Attribute Access Functions

eZCL_SendDiscoverAttributesExtendedRequest

```
teZCL_Status eZCL_SendDiscoverAttributesExtendedRequest(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    uint16 u16ClusterId,
    bool_t bDirectionIsServerToClient,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint16 u16AttributeId,
    bool_t bIsManufacturerSpecific,
    uint16 u16ManufacturerCode,
    uint8 u8MaximumNumberOfIdentifiers);
```

Description This function can be used to send a 'discover attributes extended' request to a cluster (normally a cluster server) on a remote device. The range of attributes of interest (within the standard set of cluster attributes) must be defined by specifying the identifier of the 'start' attribute and the number of attributes in the range. The function returns immediately and the results of the request are later received in a 'discover attributes extended' response.

Note: An 'extended' attribute discovery is similar to a normal attribute discovery except the accessibility of each attribute is additionally indicated as being 'read', 'write' or 'reportable'.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

On receiving the 'discover attributes extended' response, the event

E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_EXTENDED_RESPONSE

is generated for each attribute reported in the response. Therefore, multiple events normally result from a single function call ('discover attributes extended' request). Within this event, the details of the reported attribute are contained in a structure of the type tsZCL_AttributeDiscoveryExtendedResponse (see Section6.1.11).

Following the event for the final attribute reported, the event

E_ZCL_CBET_DISCOVER_ATTRIBUTES_EXTENDED_RESPONSE

is generated to indicate that all attributes from the discover attributes extended response have been reported.

Extended attribute discovery is fully described in *Appendix C.*

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent
- *u16ClusterId*: Identifier of the cluster to be queried (see the macros section in the cluster header file): :
- *bDirectionIsServerToClient*: Direction of request:

  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u16AttributeId*: Identifier of 'start' attribute of interest
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee-defined cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)
- *u8MaximumNumberOfIdentifiers*: Number of attributes in attribute range of interest (maximum number of attributes to report in response)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL

Parent topic: Attribute Access Functions

eZCL_SendConfigureReportingCommand

```
teZCL_Status eZCL_SendConfigureReportingCommand(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    uint16 u16ClusterId,
    bool_t bDirectionIsServerToClient,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint8 u8NumberOfAttributesInRequest,
    bool_t bIsManufacturerSpecific,
    uint16 u16ManufacturerCode,
    tsZCL_AttributeReportingConfigurationRecord
    *psAttributeReportingConfigurationRecord);
```

Description This function can be used on a cluster client to send a 'configure reporting' command to a cluster server, in order to request automatic reporting to be configured for a set of attributes. The configuration information is provided to the function in an array of structures, where each structure contains the configuration data for a single attribute. The function will return immediately and the results of the request will later be received in a 'configure reporting' response.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing

an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

On receiving the 'configure reporting' response, the event

E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE_RESPONSE

is generated for each attribute in the response. Therefore, multiple events will normally result from a single function call ('configure reporting' command). Following the event for the final attribute, the event

E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE_RESPONSE

is generated to indicate that the configuration outcomes for all the attributes from the 'configure reporting' command have been reported.

Note: In order for automatic reporting to be successfully configured for an attribute using this function, the 'reportable flag' for the attribute must have been set on the cluster server using the function **eZCL_SetReportableFlag()**.

Attribute reporting is fully described in *Appendix B.*

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent
- *u16ClusterId*: Identifier of the cluster to be configured (see the macros section in the cluster header file)
- *bDirectionIsServerToClient*: Direction of request:
  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request would be sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8NumberOfAttributesInRequest*: Number of attributes for which reporting is to be configured as a result of the request
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes. This code is set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE
- *psAttributeReportingConfigurationRecord*: Pointer to array of structures, where each structure contains the attributing reporting configuration data for a single attribute (see Section 6.1.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE

- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_FAIL

Parent topic: Attribute Access Functions

eZCL_SendReadReportingConfigurationCommand

```
teZCL_Status eZCL_SendReadReportingConfigurationCommand(
uint8 u8SourceEndPointId,
uint8 u8DestinationEndPointId,
uint16 u16ClusterId,
bool_t bDirectionIsServerToClient,
tsZCL_Address *psDestinationAddress,
uint8 *pu8TransactionSequenceNumber,
uint8 u8NumberOfAttributesInRequest,
bool_t bIsManufacturerSpecific,
uint16 u16ManufacturerCode,
tsZCL_AttributeReadReportingConfigurationRecord
*psAttributeReadReportingConfigurationRecord);
```

Description This function can be used on a cluster client to send a 'read reporting configuration' command to a cluster server, in order to request the attribute reporting configuration data for a set of attributes. For each attribute, configuration data can be requested relating to either sending or receiving an attribute report. The required configuration data is specified to the function in an array of structures, where each structure contains the requirements for a single attribute. The function will return immediately and the results of the request will later be received in a 'read reporting configuration' response.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

On receiving the 'read reporting configuration' response, the event

E_ZCL_CBET_REPORT_READ_INDIVIDUAL_ATTRIBUTE_CONFIGURATION_RESPONSE

is generated for each attribute in the response. Therefore, multiple events will normally result from a single function call ('read reporting configuration' command). Following the event for the final attribute reported, the event

E_ZCL_CBET_REPORT_READ_ATTRIBUTE_CONFIGURATION_RESPONSE

is generated to indicate that the configuration outcomes for all the attributes from the 'configure reporting' command have been reported.

Attribute reporting is fully described in *Appendix B.*

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent.
- *u16ClusterId*: containing the attributes (see the macros section in the cluster header file)
- *bDirectionIsServerToClient*: Direction of request:
  - TRUE: Cluster server to client

  - FALSE: Cluster client to server
- *psDestinationAddress*Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8NumberOfAttributesInRequest*: Number of attributes for which reporting is to be configured as a result of the request
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)
- *psAttributeReportingConfigurationRecord*: Pointer to an array of structures, where each structure indicates the required configuration data for a single attribute (see Section 6.1.7)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_FAIL

Parent topic: Attribute Access Functions

eZCL_ReportAllAttributes

```
teZCL_Status eZCL_ReportAllAttributes(
    tsZCL_Address *psDestinationAddress,
    uint16 u16ClusterID,
    uint8 u8SrcEndPoint,
    uint8 u8DestEndPoint,
    PDUM_thAPduInstance hAPduInst);
```

Description This function can be used on the cluster server to issue an attribute report for all the reportable attributes on the server. Only the standard attributes are reported - this does not include manufacturer-specific attributes.

Use of this function requires no special configuration on the cluster server. However, the target client must be enabled to receive attribute reports (via the compile-time option ZCL_ATTRIBUTE_REPORTING_CLIENT_SUPPORTED - see Appendix B.3.1).

After this function is called and before the attribute report is sent, the event E_ZCL_CBET_REPORT_REQUEST is generated on the server, allowing the application to update the attribute values in the shared structure, if required.

Attribute reporting is fully described in *Appendix B.*

Parameters

- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the attribute report is sent
- *u16ClusterID*: Identifier of the cluster containing the attributes to be reported (see the macros section in the cluster header file)
- *u8SrcEndPoint*: Number of endpoint on server from which attribute report is sent
- *u8DestEndPoint*: Number of endpoint on target client to which attribute report is sent
- *hAPduInst*: Handle of APDU instance that will contain the attribute report

Returns

- E_ZCL_SUCCESS

Parent topic: Attribute Access Functions

eZCL_ReportAttribute

```
teZCL_Status eZCL_ReportAttribute(
    tsZCL_Address *psDestinationAddress,
    uint16 u16ClusterID,
    uint16 u16AttributeID,
    uint8 u8SrcEndPoint,
    uint8 u8DestEndPoint,
    PDUM_thAPduInstance hAPduInst);
```

Description This function can be used on the cluster server to issue an attribute report for an individual reportable attribute on the server. Only a standard attribute can be reported - a manufacturer-specific attribute cannot be reported.

Use of this function requires no special configuration on the cluster server but the target client must be enabled to receive attribute reports (via the compile-time option ZCL_ATTRIBUTE_REPORTING_CLIENT_SUPPORTED - see Appendix B.3.1).

After this function has been called and before the attribute report is sent, the event E_ZCL_CBET_REPORT_REQUEST is generated on the server, allowing the application to update the attribute value in the shared structure, if required.

Attribute reporting is fully described in *Appendix B.*

Parameters

- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the attribute report is sent
- *u16ClusterID*: Identifier of the cluster containing the attribute to be reported (see the macros section in the cluster header file)
- *u16AttributeID*: Identifier of the attribute to be reported
- *u8SrcEndPoint*: Number of endpoint on server from which attribute report is sent
- *u8DestEndPoint*: Target client to which attribute report is sent
- *hAPduInst*: Handle of APDU instance that contains the attribute report

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND
- E_ZCL_ERR_ATTRIBUTE_NOT_REPORTABLE
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_CLUSTER_NOT_FOUND

Parent topic: Attribute Access Functions

eZCL_CreateLocalReport

```
teZCL_Status eZCL_CreateLocalReport(
    uint8 u8SourceEndPointId,
    uint16 u16ClusterId,
    bool_t bManufacturerSpecific,
    bool_t bIsServerAttribute,
    tsZCL_AttributeReportingConfigurationRecord
*psAttributeReportingConfigurationRecord);
```

Description This function can be used on a cluster server during a 'cold start' to register attribute reporting configuration data (with the ZCL) that has been retrieved from Non-Volatile Memory (NVM) using the Non-Volatile Memory Manager (NVM). Each call of the function registers the Attribute Reporting Configuration Record for a single attribute. This configuration record is supplied to the function in a structure that has been populated using the NVM. The function should only be called after the ZCL has been initialized. Following this function call, automatic attribute reporting can resume for the relevant attribute (for example, following a power loss or device reset).

The function must not be called for attributes that have not been configured for automatic attribute reporting. For example, it must not be used for attributes for which the maximum reporting interval is set to REPORTING_MAXIMUM_TURNED_OFF).

Attribute reporting is fully described in *Appendix B.*

Parameters

- *u8SourceEndPointId*: Number of endpoint on which the relevant cluster is located
- *u16ClusterId*: Identifier of the cluster containing the attribute for which retrieved attribute reporting configuration data is to be registered (see the macros section in the cluster header file)
- *bManufacturerSpecific*: Indicates whether attribute is manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attribute is manufacturer-specific
  - FALSE: Attribute is from ZigBee cluster
- *bIsServerAttribute*: Indicates whether the attribute is located on the cluster server (or client):
  - TRUE: Attribute is on cluster server
  - FALSE: Attribute is on cluster client

- *psAttributeReportingConfigurationRecord*: Pointer to structure (see Section 6.1.5) containing the reporting configuration data for the attribute

Returns

- E_ZCL_SUCCESS

Parent topic: Attribute Access Functions

eZCL_SetReportableFlag

```
teZCL_Status eZCL_SetReportableFlag(
    uint8 u8SrcEndPoint,
    uint16 u16ClusterID,
    bool bIsServerClusterInstance,
    bool bIsManufacturerSpecific,
    uint16 u16AttributeId);
```

Description This function can be used on a cluster server to set (to '1') the 'reportable flag' E_ZCL_AF_RP for an attribute. Setting this flag configures the attribute to be potentially reportable, allowing automatic reporting to be configured and implemented for the attribute. It will also allow the attribute to be reported as a result of a call to **eZCL_ReportAllAttributes()**.

The cluster on which the attribute resides must be specified. The flag will be set for the specified attribute on all endpoints, but a single endpoint must be nominated which will be used to search for the attribute definition and to check that the specified cluster has been registered with the ZCL.

Attribute reporting is fully described in *Appendix B.*

Parameters

- *u8SourceEndPointId*: Number of endpoint to be used to search for the attribute definition and to check the cluster
- *u16ClusterId*: Identifier of the cluster containing the attribute for which the flag is to be set: (see the macros section in the cluster header file)
- *bIsServerClusterInstance*: Type of cluster instance to be set:
  - TRUE: Cluster Server
  - FALSE: Cluster Client
- *bIsManufacturerSpecific*: Indicates whether attribute is manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attribute is manufacturer-specific
  - FALSE: Attribute is from ZigBee cluster:
- *u16AttributeId*: Identifier of attribute for which the flag is to be set

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_EP_RANGE

Parent topic: Attribute Access Functions

vZCL_SetDefaultReporting

```
void vZCL_SetDefaultReporting(
    tsZCL_ClusterInstance *psClusterInstance);
```

Description This function can be used on a cluster server to enable 'default reporting' for those attributes that are reportable. It should be called immediately after the cluster instance has been created.

The function checks which attributes are potentially reportable - that is, which attributes have the 'reportable flag' E_ZCL_AF_RP set. It then sets the 'default reporting flag' E_ZCL_ACF_RP for these attributes.

Note: The flag E_ZCL_AF_RP can be set for an attribute in the attribute definition or through a call to the function **eZCL_SetReportableFlag()**.

Attribute reporting is fully described in *Appendix B.*

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance for which default reporting is to be enabled (see Section 6.1.16).

Returns

- None

Parent topic: Attribute Access Functions

eZCL_HandleReadAttributesResponse

```
teZCL_Status eZCL_HandleReadAttributesResponse(
    tsZCL_CallBackEvent *psEvent,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used to examine the response to a 'read attributes' request for a remote cluster and determine whether the response is complete - that is, whether the 'read attributes' response contains all the relevant attribute values (it may be incomplete if the returned data is too large to fit into a single APDU).

eZCL_HandleReadAttributesResponse() should normally be included in the user-defined callback function that is invoked on generation of the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE. The callback function must pass the generated event into **eZCL_HandleReadAttributesResponse()**.

If the 'read attributes' response is not complete, the function will re-send 'read attributes' requests until all relevant attribute values have been received.

You are also required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *psEvent* Pointer to generated event of the type E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_CLUSTER_ID_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTE_WO
- E_ZCL_ERR_ATTRIBUTES_ACCESS
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_PARAMETER_RANGE

Parent topic: Attribute Access Functions

eZCL_ReadLocalAttributeValue

```
ZPS_teStatus eZCL_ReadLocalAttributeValue(
    uint8 u8SourceEndPointId,
    uint16 u16ClusterId,
    bool bIsServerClusterInstance,
    bool bIsManufacturerSpecific,
    bool_t bIsClientAttribute,
    uint16 u16AttributeId,
    void *pvAttributeValue);
```

Description This function can be used to read a local attribute value of the specified cluster on the specified endpoint. Before reading the attribute value, the function checks that the attribute and cluster actually reside on the endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint on which the read will be performed
- *u16ClusterId* Identifier of the cluster to be read (see the macros section in the cluster header file)
- *bIsServerClusterInstance* Type of cluster instance to be read:
- TRUE: Cluster server
- FALSE: Cluster client
- *bIsManufacturerSpecific* Indicates whether attribute is manufacturer-specific or as defined in relevant ZigBee cluster:
- TRUE: Attribute is manufacturer-specific

- FALSE: Attribute is from ZigBee cluster
- *blsClientAttribute* Type of attribute to be read (client or server):
- TRUE: Client attribute
- FALSE: Server attribute
- *u16AttributeId* Identifier of the attribute to be read
- *pvAttributeValue* Pointer to location to receive the read attribute value

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_CLUSTER_ID_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTE_WO
- E_ZCL_ERR_ATTRIBUTES_ACCESS
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_PARAMETER_RANGE

Parent topic: Attribute Access Functions

eZCL_WriteLocalAttributeValue

```
ZPS_teStatus eZCL_WriteLocalAttributeValue(
    uint8 u8SourceEndPointId,
    uint16 u16ClusterId,
    bool bIsServerClusterInstance,
    bool bIsManufacturerSpecific,
    bool_t blsClientAttribute,
    uint16 u16AttributeId,
    void *pvAttributeValue);
```

Description This function writes a value to a local attribute value of the specified cluster on the specified endpoint. Before writing the attribute value, the function checks that the attribute and cluster actually reside on the endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint on which the write is performed
- *u16ClusterId*: Identifier of the cluster to be written to (see the macros section in the cluster header file):
- *bIsServerClusterInstance*: Type of cluster instance to be written to:
  - TRUE: Cluster server
  - FALSE: Cluster client

- *bIsManufacturerSpecific*: Indicates whether attribute is manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attribute is manufacturer-specific
  - FALSE: Attribute is from ZigBee cluster
- *blsClientAttribute*: Type of attribute to be written to (client or server):
  - TRUE: Client attribute
  - FALSE: Server attribute
- *u16AttributeId*: Identifier of the attribute to be written to
- *pvAttributeValue*: Pointer to location containing the attribute value to be written

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_CLUSTER_ID_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTE_WO
- E_ZCL_ERR_ATTRIBUTES_ACCESS
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_PARAMETER_RANGE

Parent topic: Attribute Access Functions

eZCL_OverrideClusterControlFlags

```
teZCL_Status eZCL_OverrideClusterControlFlags(
uint8 u8SrcEndpoint,
uint16 u16ClusterId,
bool bIsServerClusterInstance,
uint8 u8ClusterControlFlags);
```

Description This function can be used to over-ride the control flag setting for the specified cluster (it can be used for any cluster). If required, this function can be called immediately after the relevant endpoint registration function (for example, for a Light Sensor device, **eHA_RegisterLightSensorEndPoint()**) or at any subsequent point in the application.

In particular, this function can be used by the application to change the default security level for a cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint on which the control flag is to be overridden
- *u16ClusterId*: Identifier of the cluster to have control flag over-ridden (see the macros section in the cluster header file):

- *bIsServerClusterInstance*: Type of cluster instance:
  - TRUE: Cluster server
  - FALSE: Cluster client
- *u8ClusterControlFlags*: Value to be written to control flag, one of:
  - E_ZCL_SECURITY_NETWORK
  - E_ZCL_SECURITY_APPLINK

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Attribute Access Functions

eZCL_SetSupportedSecurity

```
teZCL_Status eZCL_SetSupportedSecurity(
    teZCL_ZCLSendSecurity eSecuritySupported);
```

Description This function can be used to set the security level for future transmissions from the local device. The possible levels are:

- Application-level security, which uses an application link key that is unique to the pair of nodes in communication
- Network-level security, which uses a network key that is shared by the whole network

By default, application-level security is enabled. In practice, this function can be used to disable application-level security on the local device so that the device sends all future communications with only network-level security. This is useful when transmitted packets need to be easily accessed. For example, it can be used during over-air tests performed using a packet sniffer.

Parameters

- eSecuritySupportedRequired level of security, one of:
  - E_ZCL_SECURITY_NETWORK - network-level security
  - E_ZCL_SECURITY_APPLINK - application-level security

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_RANGE

Parent topic: Attribute Access Functions

Parent topic:*ZCL Functions*

Command Discovery Functions The following functions are provided in the ZCL for performing command discovery:

- eZCL_SendDiscoverCommandReceivedRequest
- eZCL_SendDiscoverCommandGeneratedRequest

Note: In order to use these functions, Command Discovery must be enabled in the compile-time options. For more details, refer to the introduction to Command Discovery in Section 2.9.

eZCL_SendDiscoverCommandReceivedRequest

```
teZCL_Status eZCL_SendDiscoverCommandReceivedRequest(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    uint16 u16ClusterId,
    bool_t bDirectionIsServerToClient,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint8 u8CommandId,
    bool_t bIsManufacturerSpecific,
    uint16 u16ManufacturerCode,
    uint8 u8MaximumNumberOfCommands);
```

Description This function sends a request to initiate a command discovery on a remote cluster instance to obtain a list of commands that can be received by the cluster instance.

Commands are represented by their Command IDs and the first Command ID from which the discovery is to start must be specified. The maximum number of commands to be reported must also be specified. This allows the function can be called multiple times to discover the commands in stages (see below).

The function also allows commands to be searched for that are associated with a particular manufacturer code. Alternatively, the manufacturer code can be searched for, along with the commands.

The target cluster returns a response containing the requested information. On receiving this response, the following events are generated on the local device:

- E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_RECEIVED_RESPONSE: This event is generated for each individual command reported in the response. The reported information is contained in a structure of the type tsZCL_CommandDiscoveryIndividualResponse (see Section 6.1.17).
- E_ZCL_CBET_DISCOVER_COMMAND_RECEIVED_RESPONSE: This event is generated after all the above individual events, in order to indicate the end of these events. The reported information is contained in a structure of the type tsZCL_CommandDiscoveryResponse (see Section 6.1.18).

The tsZCL_CommandDiscoveryResponse structure in the last event contains a flag which indicates whether there are still commands to be discovered. If this is the case, the function can be called again with a new starting point (first Command ID).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Command discovery is described in Section 2.9.

Parameters

- *u8SourceEndPointId*Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint (hosting the target cluster instance) to which the request is sent
- *u16ClusterId*: Identifier of the cluster for which a command discovery is requested
- *bDirectionIsServerToClient*: Boolean indicating the type of request in terms of source and target clusters:
- TRUE - server sending request to client
- FALSE - client sending request to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8CommandId*: Command ID which is the starting point for the command discovery
- *bIsManufacturerSpecific*: Boolean indicating whether a manufacturer code is specified in the parameter *u16ManufacturerCode* below:
  - TRUE - *u16ManufacturerCode* is used
  - FALSE - *u16ManufacturerCode* is not used
- *u16ManufacturerCode*: A manufacturer-specific code (depends on the setting of *bIsManufacturerSpecific* above). 0xFFFF is a wildcard value indicating that the manufacturer code should be discovered along with the commands
- *u8MaximumNumberOfCommands*: Maximum number of commands to be discovered

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Command Discovery Functions

eZCL_SendDiscoverCommandGeneratedRequest

```
teZCL_Status eZCL_SendDiscoverCommandGeneratedRequest(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    uint16 u16ClusterId,
    bool_t bDirectionIsServerToClient,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint8 u8CommandId,
    bool_t bIsManufacturerSpecific,
    uint16 u16ManufacturerCode,
    uint8 u8MaximumNumberOfCommands);
```

Description This function sends a request to initiate a command discovery on a remote cluster instance to obtain a list of commands that can be generated by the cluster instance.

Commands are represented by their Command IDs and the first Command ID from which the discovery is to start must be specified. The maximum number of commands to be reported must also be specified. This allows the function can be called multiple times to discover the commands in several stages.

The function also allows commands to be searched for that are associated with a particular manufacturer code. Alternatively, the manufacturer code can be searched for, along with the commands.

The target cluster returns a response containing the requested information. On receiving this response, the following events are generated on the local device:

- E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_GENERATED_RESPONSE: This event is generated for each individual command reported in the response. The reported information is contained in a structure of the type tsZCL_CommandDiscoveryIndividualResponse (see Section 6.1.17).
- E_ZCL_CBET_DISCOVER_COMMAND_GENERATED_RESPONSE: This event is generated after all the above individual events, in order to indicate the end of these events. The reported information is contained in a structure of the type tsZCL_CommandDiscoveryResponse (see Section 6.1.18).

The tsZCL_CommandDiscoveryResponse structure in the last event contains a flag which indicates whether there are still commands to be discovered. If this is the case, the function can be called again with a new starting point (first Command ID).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Command discovery is described in Section 2.9.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint (hosting the target cluster instance) to which the request is sent
- *u16ClusterId*: Identifier of the cluster for which a command discovery is requested
- *bDirectionIsServerToClient*: Boolean indicating the type of request in terms of source and target clusters:
  - TRUE - server sending request to client
  - FALSE - client sending request to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8CommandId*: Command ID which is the starting point for the command discovery
- *bIsManufacturerSpecific*: Boolean indicating whether a manufacturer code is specified in the parameter *u16ManufacturerCode* below:
  - TRUE - *u16ManufacturerCode*is used
  - FALSE - *u16ManufacturerCode*is not used

- *u16ManufacturerCode*: A manufacturer-specific code (depends on the setting of *bIsManufacturerSpecific* above). 0xFFFF is a wildcard value indicating that the manufacturer code should be discovered along with the commands
- *u8MaximumNumberOfCommands*: Maximum number of commands to be discovered

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL

Parent topic: Command Discovery Functions

Parent topic:*ZCL Functions*

ZCL Structures

This chapter details the structures that are not specific to any particular ZCL cluster.

Note: Cluster-specific structures are detailed in the chapters for the respective clusters.

General Structures

tsZCL_EndPointDefinition

This structure defines the endpoint for an application:

```
struct tsZCL_EndPointDefinition
{
    uint8                        u8EndPointNumber;
    uint16                       u16ManufacturerCode;
    uint16                       u16ProfileEnum;
    bool_t                       bIsManufacturerSpecificProfile;
    uint16                       u16NumberOfClusters;
    tsZCL_ClusterInstance        *psClusterInstance;
    bool_t                       bDisableDefaultResponse;
    tfpZCL_ZCLCallBackFunction   pCallBackFunctions;
};
```

Where:

- u8EndPointNumber is the endpoint number between 1 and 240 (0 is reserved)
- u16ManufacturerCode is the manufacturer code (only valid when bIsManufacturerSpecificProfile is set to TRUE)
- u16ProfileEnum is the ZigBee application profile ID
- bIsManufacturerSpecificProfile indicates whether the application profile is proprietary (TRUE) or from the ZigBee Alliance (FALSE)
- u16NumberOfClusters is the number of clusters on the endpoint
- psClusterInstance is a pointer to an array of cluster instance structures
- bDisableDefaultResponse can be used to disable the requirement for default responses to be returned for commands sent from the endpoint (TRUE=disable, FALSE=enable)
- pCallBackFunctions is a pointer to the callback functions for the endpoint

Parent topic: General Structures

tsZCL_ClusterDefinition This structure defines a cluster used on a device:

```
typedef struct
{
    uint16                    u16ClusterEnum;
    bool_t                    bIsManufacturerSpecificCluster;
    uint8                     u8ClusterControlFlags;
    uint16                    u16NumberOfAttributes;
    tsZCL_AttributeDefinition  *psAttributeDefinition;
    tsZCL_SceneExtensionTable  *psSceneExtensionTable;
#ifdef ZCL_COMMAND_DISCOVERY_SUPPORTED
    uint8                     u8NumberOfCommands;
    tsZCL_CommandDefinition    *psCommandDefinition;
#endif
}  tsZCL_ClusterDefinition;
```

Where:

- u16ClusterEnum is the Cluster ID.
- bIsManufacturerSpecificCluster indicates whether the cluster is specific to a manufacturer (proprietary):
  - TRUE - proprietary cluster
  - FALSE - ZigBee cluster

u8ClusterControlFlags is a bitmap containing control bits in two parts, as follows:

Bits	Description	Values
0 - 3	Type of security	Indicates the type of security key used via one of the following teZCL_ZCLSendSecurity enumerations (see Section 7.1.6):

Bits	Description	Values
0 - 3	Type of security	Indicates the type of security key used via one of the following teZCL_ZCLSend Security enumerations (see Section 7.1.6): E_ZCL_SECURITY_NETWORK, E_ZCL_SECURITY_APPLINK• E_ZCL_SECURITY_TEMP_APPL option is for internal use only)
4 - 7	Cluster mirror	Used internally to indicate whether the cluster is mirrored, as follows:• 0000b - Not mirrored,1000b - Mirrored, All other values are reserved

- u16NumberOfAttributes indicates the number of attributes in the cluster.
- psAttributeDefinition is a pointer to an array of attribute definition structures - see Section 6.1.3.
- psSceneExtensionTable is a pointer to a structure containing a Scene Extension table - see Section 6.1.20.

- The following optional pair of fields are related to the Command Discovery feature (see Section 2.9):
  - u8NumberOfCommands is the number of supported commands in the Command Definition table (see below).
  - psCommandDefinition is a pointer to a Command Definition table which contains a list of the commands supported by the cluster - each entry of the table contains the details of a supported command in a tsZCL_CommandDefinition structure (see Section 6.1.19).

Parent topic: General Structures

tsZCL_AttributeDefinition This structure defines an attribute used in a cluster:

```
struct tsZCL_AttributeDefinition
{
    uint16                  u16AttributeEnum;
    uint8                   u8AttributeFlags;
    teZCL_ZCLAttributeType  eAttributeDataType;
    uint16                  u16OffsetFromStructBase;
    uint16                  u16AttributeArrayLength;
};
```

Where:

- u16AttributeEnum is the Attribute ID.
- u8AttributeFlags is a 5-bit bitmap indicating the accessibility of the attribute (for details of the access types, refer to Section 2.3.1) - a bit is set to '1' if the corresponding access type is supported, as follows:

Bit	Access Type
0	Read
1	Write
2	Reportable
3	Scene
4	Global
5-7	Reserved

- eAttributeDataType is the data type of the attribute - see Section 7.1.3.
- u16O``ffsetFromStructBase is the offset of the attribute's location from the start of the cluster.
- u16AttributeArrayLength is the number of consecutive attributes of the same type.

Parent topic: General Structures

tsZCL_Address This structure is used to specify the addressing mode and address for a communication with a remote node:

```
typedef struct PACK
{
  teZCL_AddressMode           eAddressMode;
  union {
        zuint16                u16GroupAddress;
        zuint16                u16DestinationAddress;
        zuint64                u64DestinationAddress;
        teAplAfBroadcastMode   eBroadcastMode;
    } uAddress;
```

(continues on next page)

(continued from previous page)

```
} tsZCL_Address;
```

Where:

- eAddressMode is the addressing mode to be used (see Section 7.1.1).
- uAddress is a union containing the necessary address information (only one of the following must be set, depending on the addressing mode selected):
  - u16GroupAddress is the 16-bit group address for the target nodes.
  - u16DestinationAddress is the 16-bit network address of the target.
  - u64DestinationAddress is the 64-bit IEEE/MAC address of the target.
  - eBroadcastMode is the required broadcast mode (see Section 7.1.2).

Parent topic: General Structures

tsZCL_AttributeReportingConfigurationRecord This structure contains the configuration record for automatic reporting of an attribute.

```
typedef struct
{
    uint8                    u8DirectionIsReceived;
    teZCL_ZCLAttributeType           eAttributeDataType;
    uint16                    u16AttributeEnum;
    uint16                    u16MinimumReportingInterval;
    uint16                    u16MaximumReportingInterval;
    uint16                    u16TimeoutPeriodField;
    tuZCL_AttributeReportable        uAttributeReportableChange;
} tsZCL_AttributeReportingConfigurationRecord;
```

Where:

- u8DirectionIsReceived indicates whether the record configures how attribute reports can be received or sent:
  - 0x00: Configures how attribute reports are sent by the server - the following fields are included in the message payload:

    eAttributeDataType, u16MinimumReportingInterval, u16MaximumReportingInterval, uAttributeReportableChange
  - 0x01: Configures how attribute reports are received by the client - u16TimeoutPeriodField is included in the message payload.
- eAttributeDataType indicates the data type of the attribute.
- u16AttributeEnum is the identifier of the attribute to which the configuration record relates.
- u``16MinimumReportingInterval is the minimum time-interval, in seconds, between consecutive reports for the attribute - the value 0x0000 indicates no minimum (REPORTING_MINIMUM_LIMIT_NONE).
- u16MaximumReportingInterval is the time-interval, in seconds, between consecutive reports for periodic reporting - the following special values can also be set:
  - 0x0000 indicates that periodic reporting is to be disabled for the attribute (REPORTING_MAXIMUM_PERIODIC_TURNED_OFF).
  - 0xFFFF indicates that automatic reporting is to be completely disabled for the attribute (REPORTING_MAXIMUM_TURNED_OFF).

- u16TimeoutPeriodField is the timeout value, in seconds, for an attribute report - if the time elapsed since the last report exceeds this value (without receiving another report), it may be assumed that there is a problem with the attribute reporting - the value 0x0000 indicates that no timeout will be applied (REPORTS_OF_ATTRIBUTE_NOT_SUBJECT_TO_TIMEOUT).
- uAttributeReportableChange is the minimum change in the attribute value that causes an attribute report to be issued.

Note: For successful attribute reporting, the timeout on the receiving client must be set to a higher value than the maximum reporting interval for the attribute on the sending server.

Parent topic: General Structures

tsZCL_AttributeReportingConfigurationResponse This structure contains information from a 'configure reporting' response.

```
typedef struct
{
    teZCL_CommandStatus  eCommandStatus;
    tsZCL_AttributeReportingConfigurationRecord
                sAttributeReportingConfigurationRecord;
}tsZCL_AttributeReportingConfigurationResponse;
```

Where:

- eCommandStatus is an enumeration representing the status from the response (see Section 7.1.4).
- sAttributeReportingConfigurationRecord is a configuration record structure (see Section 6.1.5), but only the fields u16AttributeEnum and u8DirectionIsReceived are used in the response.

Parent topic: General Structures

tsZCL_AttributeReadReportingConfigurationRecord This structure contains the details of a reporting configuration query for one attribute, to be included in a 'read reporting configuration' command:

```
typedef struct
{
    uint8       u8DirectionIsReceived;
    uint16      u16AttributeEnum;
} tsZCL_AttributeReadReportingConfigurationRecord;
```

Where:

- u8DirectionIsReceived specifies whether the required reporting configuration information details how the attribute reports are received or sent.
  - 0x00: Specifies that required information details how a report is sent by the server.
  - 0x01: Specifies that required information details how a report is received by the client.
- u16AttributeEnum is the identifier of the attribute to which the required reporting configuration information relates.

Parent topic: General Structures

tsZCL_IndividualAttributesResponse This structure is contained in a ZCL event of type E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE (see Section):

```
typedef struct PACK {
    uint16              u16AttributeEnum;
    teZCL_ZCLAttributeType        eAttributeDataType;
    teZCL_CommandStatus          eAttributeStatus;
    void               *pvAttributeData;
} tsZCL_IndividualAttributesResponse;
```

Where:

- u16AttributeEnum identifies the attribute that has been read (the relevant enumerations are listed in the 'Enumerations' section of each cluster-specific chapter).
- eAttributeDataType is the ZCL data type of the read attribute (see Section 7.1.3).
- eAttributeStatus is the status of the read operation (0x00 for success or an error code - see Section 7.1.4 for enumerations).
- pvAttributeData is a pointer to the read attribute data which (if the read was successful) has been inserted by the ZCL into the shared device structure.

The above structure is contained in the tsZCL_CallBackEvent event structure, detailed in Section 6.2, when the field eEventType is set to E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE.

Parent topic: General Structures

tsZCL_DefaultResponse This structure is contained in a ZCL event of type E_ZCL_CBET_DEFAULT_RESPONSE (see Section):

```
typedef struct PACK {
    uint8  u8CommandId;
    uint8  u8StatusCode;
} tsZCL_DefaultResponse;
```

Where:

- u8CommandId is the ZCL identifier of the command that triggered the default response message
- u8StatusCode is the status code from the default response message (0x00 for OK or an error code defined in the ZCL Specification - see Section 4.2)

The above structure is contained in the tsZCL_CallBackEvent event structure, detailed in Section 6.2, when the field eEventType is set to E_ZCL_CBET_DEFAULT_RESPONSE.

Parent topic: General Structures

tsZCL_AttributeDiscoveryResponse This structure contains details of an attribute reported in a 'discover attributes' response. It is contained in a ZCL event of type E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_RESPONSE.

```
typedef struct
{
    bool_t                bDiscoveryComplete;
    uint16               u16AttributeEnum;
     teZCL_ZCLAttributeType          eAttributeDataType;
} tsZCL_AttributeDiscoveryResponse;
```

where:

- bDiscoveryComplete indicates whether this is the final attribute from a 'discover attributes' to be reported:
  - TRUE - final attribute
  - FALSE - not final attribute
- u16AttributeEnum is the identifier of the attribute being reported
- eAttributeDataType indicates the data type of the attribute being reported (see Section 7.1.3)

The above structure is contained in the tsZCL_CallBackEvent event structure, detailed in Section 6.2, when the field eEventType is set to E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_RESPONSE.

Parent topic: General Structures

tsZCL_AttributeDiscoveryExtendedResponse This structure contains details of an attribute reported in a 'discover attributes extended' response. It is contained in a ZCL event of type E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_EXTENDED_RESPONSE.

```
typedef struct
{
    bool_t                  bDiscoveryComplete;
    uint16                  u16AttributeEnum;
    teZCL_ZCLAttributeType  eAttributeDataType;
    uint8                   u8AttributeFlags;
}tsZCL_AttributeDiscoveryExtendedResponse;
```

where:

- bDiscoveryComplete indicates whether this is the final attribute from a 'discover attributes' to be reported:
  - TRUE - final attribute
  - FALSE - not final attribute
- u16AttributeEnum is the identifier of the attribute being reported
- eAttributeDataType indicates the data type of the attribute being reported (see Section 7.1.3)
- u8AttributeFlags is a 5-bit bitmap indicating the accessibility of the reported attribute (for details of the access types, refer to Section 2.3.1) - a bit is set to '1' if the corresponding access type is supported, as follows:

Bit	Access Type
0	Read
1	Write
2	Reportable
3	Scene
4	Global
5-7	Reserved

The above structure is contained in the tsZCL_CallBackEvent event structure, detailed in Section 6.2, when the field eEventType is set to E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_EXTENDED_RESPONSE.

Parent topic: General Structures

tsZCL_ReportAttributeMirror This structure contains information relating to a report attribute command:

```
typedef struct
{
    uint8                    u8DestinationEndPoint;
    uint16                   u16ClusterId;
    uint64                   u64RemoteIeeeAddress;
    teZCL_ReportAttributeStatus eStatus;
}tsZCL_ReportAttributeMirror;
```

where:

- u8DestinationEndPoint is the number of target endpoint for the attribute report (this is the endpoint on which the mirror for the device resides)
- u16ClusterId is the ID of the cluster for which information is to be mirrored
- u64RemoteIeeeAddress is the IEEE/MAC address of the target device for the attribute report (which contains the mirror for the device)
- eStatus indicates the status of the attribute report (see Section 7.1.5)

Parent topic: General Structures

tsZCL_OctetString This structure contains information on a ZCL octet (byte) string. This string is of the format:

|**Octet Count, N**

(1 octet)

|**Data**

(N octets)

|

which contains N+1 octets, where the leading octet indicates the number of octets (N) of data in the remainder of the string (valid values are from 0x00 to 0xFE).

The tsZCL_OctetString structure incorporates this information as follows:

```
typedef struct
{
    uint8    u8MaxLength;
    uint8    u8Length;
    uint8   *pu8Data;
} tsZCL_OctetString;
```

Where:

- u8MaxLength is the maximum number of data octets in an octet string
- u8Length is the actual number of data octets (N) in this octet string
- pu8Data is a pointer to the first data octet of this string

Note that there is also a tsZCL_LongOctetString structure in which the octet count (N) is represented by two octets, thus allowing double the number of data octets.

Parent topic: General Structures

tsZCL_CharacterString This structure contains information on a ZCL character string. This string is of the format:

Character Data Length, L	Character Data
(1 byte)	(L bytes)

which contains L+1 bytes, where the leading byte indicates the number of bytes (L) of character data in the remainder of the string (valid values are from 0x00 to 0xFE). This value represents the number of characters in the string only if the character set used encodes each character using one byte (this is the case for ISO 646 ASCII but not in all character sets, for example, UTF8).

The tsZCL_CharacterString structure incorporates this information as follows:

```
typedef struct
{
    uint8     u8MaxLength;
    uint8     u8Length;
    uint8    *pu8Data;
} tsZCL_CharacterString;
```

where:

- u8MaxLength is the maximum number of character data bytes
- u8Length is the actual number of character data bytes (L) in this string
- pu8Data is a pointer to the first character data byte of this string

The string is not null-terminated and may therefore contain null characters mid-string.

Note that there is also a sZCL_LongCharacterString structure in which the character data length (L) is represented by two bytes, thus allowing double the number of characters.

Parent topic: General Structures

tsZCL_ClusterCustomMessage This structure contains a cluster custom message:

```
typedef struct {
    uint16                  u16ClusterId;
    void                   *pvCustomData;
} tsZCL_ClusterCustomMessage;
```

Where:

- u16ClusterId is the Cluster ID.
- pvCustomData is a pointer to the start of the data contained in the message.

Parent topic: General Structures

tsZCL_ClusterInstance This structure contains information about an instance of a cluster on a device:

```
struct tsZCL_ClusterInstance
{
    bool_t                  bIsServer;
    tsZCL_ClusterDefinition    *psClusterDefinition;
```

(continues on next page)

(continued from previous page)

```
    void                    *pvEndPointSharedStructPtr;
    uint8                   *pu8AttributeControlBits;
    void                    *pvEndPointCustomStructPtr;
    tfpZCL_ZCLCustomcallCallBackFunction
                            pCustomcallCallBackFunction;
};
```

where:

- bIsServer indicates whether the cluster instance is a server or client:
  - TRUE - server
  - FALSE - client
- psClusterDefinition is a pointer to the cluster definition structure - see Section 6.1.2
- pvEndPointSharedStructPtr is a pointer to the shared device structure that contains the cluster's attributes
- pu8AttributeControlBits is a pointer to an array of bitmaps, one for each attribute in the relevant cluster - for internal cluster definition use only, array should be initialized to 0
- pvEndPointCustomStructPtr is a pointer to any custom data (only relevant to a user-defined cluster)
- pCustomcallCallBackFunction is a pointer to a custom callback function (only relevant to a user-defined cluster)

Parent topic: General Structures

tsZCL_CommandDiscoveryIndividualResponse This structure contains information about an individual command reported in a Command Discovery response (see Section 2.9).

```
typedef struct
{
    uint8   u8CommandEnum;
    uint8   u8CommandIndex;
} tsZCL_CommandDiscoveryIndividualResponse;
```

where:

- u8CommandEnum is the Command ID of the reported command
- u8CommandIndex is the index of the reported command in the response payload

The above structure is contained in the tsZCL_CallBackEvent event structure, detailed in Section 6.2, when the field eEventType is set to E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_RECEIVED_RESPONSE or E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_GENERATED_RESPONSE.

Parent topic: General Structures

tsZCL_CommandDiscoveryResponse This structure contains information about a Command Discovery response (see Section 2.9).

```
typedef struct
{
    bool_t bDiscoveryComplete;
    uint8  u8NumberOfCommands;
```

(continues on next page)

(continued from previous page)

```
} tsZCL_CommandDiscoveryResponse;
```

Where:

- bDiscoveryComplete is a Boolean flag which indicates whether the Command Discovery is complete, i.e. whether there are any commands remaining to be discovered:
  - TRUE - all commands have been discovered
  - FALSE - there are further commands to be discovered
- u8NumberOfCommands is the number of discovered commands reported in the response (the individual commands are reported in a structure of the type tsZCL_CommandDiscoveryIndividualResponse - see Section 6.1.17)

The above structure is contained in the tsZCL_CallBackEvent event structure, detailed in Section 6.2, when the field eEventType is set to E_ZCL_CBET_DISCOVER_COMMAND_RECEIVED_RESPONSE or E_ZCL_CBET_DISCOVER_COMMAND_GENERATED_RESPONSE.

Parent topic: General Structures

tsZCL_CommandDefinition This structure contains the details of a command which is supported by the cluster (and can be reported in Command Discovery).

```
struct tsZCL_CommandDefinition
{
    uint8   u8CommandEnum;
    uint8   u8CommandFlags;
};
```

Where:

- u8CommandEnum is the Command ID within the cluster
- u8CommandFlags is a bitmap containing a set of control flags, as follows:

Bits	Enumeration	Description
0	E_ZCL_CF_RX	Command is generated by the client and received by the server
1	E_ZCL_CF_TX	Command is generated by the server and received by the client
2	-	Reserved
3	E_ZCL_CF_MS	Command is manufacturer-specific
4 - 7	-	Reserved

Parent topic: General Structures

tsZCL_SceneExtensionTable This structure contains a Scenes Extension table.

```
typedef struct
{
    tfpZCL_SceneEventHandler     pSceneEventHandler;
    uint16                       u16NumberOfAttributes;
    uint16                       au16Attributes[];
} tsZCL_SceneExtensionTable;
```

Where:

- pSceneEventHandler is a pointer a Scenes event handler function
- u16NumberOfAttributes is the number of attributes in the Scene extension
- au16Attributes is an array of the attribute IDs of the attributes in the Scene extension

Parent topic: General Structures

tsZCL_WriteAttributeRecord The is structure contains the details for a 'write attribute' operation.

```
typedef struct
{
        teZCL_ZCLAttributeType   eAttributeDataType;
        uint16                   u16AttributeEnum;
        uint8                                          *pu8AttributeData;
}tsZCL_WriteAttributeRecord;
```

Where:

- eAttributeDataType is an enumeration indicating the attribute data type (for the enumerations, refer to Section 7.1.3).
- u16AttributeEnum is an enumeration for the attribute identifier (for the relevant 'Attribute ID' enumerations, refer to the 'Enumerations' section of each cluster-specific chapter).
- pu8AttributeData is a pointer to the attribute data to be written.

Parent topic: General Structures

Parent topic:*ZCL Structures*

Event Structure (tsZCL_CallBackEvent) A ZCL event must be wrapped in the following tsZCL_CallBackEvent structure before being passed into the function **vZCL_EventHandler()**:

```
typedef struct
 {
teZCL_CallBackEventType                eEventType;
uint8                                  u8TransactionSequenceNumber;
uint8                                  u8EndPoint;
teZCL_Status                           eZCL_Status;
union {
tsZCL_IndividualAttributesResponse     sIndividualAttributeResponse;
tsZCL_DefaultResponse                  sDefaultResponse;
tsZCL_TimerMessage                     sTimerMessage;
tsZCL_ClusterCustomMessage             sClusterCustomMessage;
tsZCL_AttributeReportingConfigurationRecord
                                       sAttributeReportingConfigurationRecord;
tsZCL_AttributeReportingConfigurationResponse
                                       sAttributeReportingConfigurationResponse;
tsZCL_AttributeDiscoveryResponse       sAttributeDiscoveryResponse;
tsZCL_AttributeStatusRecord            sReportingConfigurationResponse;
tsZCL_ReportAttributeMirror            sReportAttributeMirror;
uint32                                 u32TimerPeriodMs;
tsZCL_CommandDiscoveryIndividualResponse
                                       sCommandsReceivedDiscoveryIndividualResponse;
tsZCL_CommandDiscoveryResponse         sCommandsReceivedDiscoveryResponse;
tsZCL_CommandDiscoveryIndividualResponse
                                       sCommandsGeneratedDiscoveryIndividualResponse;
tsZCL_CommandDiscoveryResponse         sCommandsGeneratedDiscoveryResponse;
tsZCL_AttributeDiscoveryExtendedResponse
                                       sAttributeDiscoveryExtenedResponse;
```

(continues on next page)

(continued from previous page)

```
}uMessage ;
    ZPS_tsAfEvent *pZPSevent;
    tsZCL_ClusterInstance *psClusterInstance;
} tsZCL_CallBackEvent;
```

where

- eEventType: specifies the type of event generated - see Section 7.3.
- u8TransactionSequenceNumber is the Transaction Sequence Number (TSN) of the incoming ZCL message (if any) which triggered the ZCL event.
- u8EndPoint is the endpoint on which the ZCL message (if any) was received.
- eZCL_Status is the status of the operation that the event reports - see Section 7.2
- uMessage is a union containing information that is only valid for specific events:
  - s``I``ndividualAttributeResponse contains the response to a 'read attributes' or 'write attributes' request - see Section 6.1.8.
  - sDefaultResponse contains the response to a request (other than a read request) - see Section 6.1.9.
  - sTimerMessage contains the details of a timer event - this feature is included for future use.
  - sClusterCustomMessage contains details of a cluster custom command - see Section 6.1.15
  - sAttributeReportingConfigurationRecord contains the attribute reporting configuration data from the 'configure reporting' request for an attribute - see Section 6.1.5.
  - sAttributeReportingConfigurationResponse is reserved for future use.
  - sAttributeDiscoveryResponse contains the details of an attribute reported in a 'discover attributes' response - see Section 6.1.10.
  - sReportingConfigurationResponse is reserved for future use.
  - sReportAttributeMirror contains information on the device from which a ZCL 'report attribute' command has been received.
  - u32TimerPeriodMs contains the timed period of the millisecond timer which is enabled by the application when the event E_ZCL_CBET_ENABLE_MS_TIMER occurs.
  - s``CommandsReceivedDiscoveryIndividualResponse contains information about an individual command (that can be received) reported in a Command Discovery response - see Section 6.1.17.
  - sCommandsReceivedDiscoveryResponse contains information about a Command Discovery response which reports commands that can be received - see Section 6.1.18.
  - sCommandsGeneratedDiscoveryIndividualResponse contains information about an individual command (that can be generated) reported in a Command Discovery response - see Section 6.1.17.
  - sCommandsGeneratedDiscoveryResponse contains information about a Command Discovery response which reports commands that can be generated - see Section 6.1.18.
  - sAttributeDiscoveryExtenedResponse contains information from a Discover Attributes Extended response - see Section 6.1.11.

The remaining fields are common to more than one event type but are not valid for all events:

- pZPSevent is a pointer to the stack event (if any) that caused the ZCL event.
- psClusterInstance is a pointer to the cluster instance structure that holds the information relating to the cluster being accessed.

Parent topic:*ZCL Structures*

Enumerations and Status Codes This chapter details the enumerations and status codes provided in the NXP implementation of the ZCL or provided in the ZigBee PRO APIs and used by the ZCL.

General Enumerations

Addressing Modes (teZCL_AddressMode) The following enumerations are used to specify the addressing mode to be used in a communication with a remote node:

```
typedef enum
{
    E_ZCL_AM_BOUND,
    E_ZCL_AM_GROUP,
    E_ZCL_AM_SHORT,
    E_ZCL_AM_IEEE,
    E_ZCL_AM_BROADCAST,
    E_ZCL_AM_NO_TRANSMIT,
    E_ZCL_AM_BOUND_NO_ACK,
    E_ZCL_AM_SHORT_NO_ACK,
    E_ZCL_AM_IEEE_NO_ACK,
    E_ZCL_AM_BOUND_NON_BLOCKING,
    E_ZCL_AM_BOUND_NON_BLOCKING_NO_ACK,
    E_ZCL_AM_ENUM_END, /* enum End */
} teZCL_AddressMode;
```

The above enumerations are described in the table below.

Enumeration	Description
E_ZCL_AM_BOUND	Use one or more bound nodes/endpoints, with acknowledgments
E_ZCL_AM_GROUP	Use a pre-defined group address, with acknowledgments
E_ZCL_AM_SHORT	Use a 16-bit network address, with acknowledgments
E_ZCL_AM_IEEE	Use a 64-bit IEEE/MAC address, with acknowledgments
E_ZCL_AM_BROADCAST	Perform a broadcast (see Section 7.1.2)
E_ZCL_AM_NO_TRANSMIT	Do not transmit
E_ZCL_AM_BOUND_NO_ACK	Perform a bound transmission, with no acknowledgments
E_ZCL_AM_SHORT_NO_ACK	Perform a transmission using a 16-bit network address, with no acknowledgments
E_ZCL_AM_IEEE_NO_ACK	Perform a transmission using a 64-bit IEEE/MAC address, with no acknowledgments
E_ZCL_AM_BOUND_NON_BLOCKI	Perform a non-blocking bound transmission, with acknowledgments
E_ZCL_AM_BOUND_NON_BLOCKI	Perform a non-blocking bound transmission, with no acknowledgments

The required addressing mode is specified in the structure tsZCL_Address (see Section 6.1.4).

Parent topic: General Enumerations

Broadcast Modes (ZPS_teAplAfBroadcastMode) The following enumerations are used to specify the type of broadcast (when the addressing mode for a communication has been set to E_ZCL_AM_BROADCAST (see Section 7.1.1)):

```
typedef enum
{
    ZPS_E_APL_AF_BROADCAST_ALL,
    ZPS_E_APL_AF_BROADCAST_RX_ON,
    ZPS_E_APL_AF_BROADCAST_ZC_ZR
} ZPS_teAplAfBroadcastMode;
```

The above enumerations are described in the table below.

Enumeration	Description
ZPS_E_APL_AF_BROADCAST	All End Devices
ZPS_E_APL_AF_BROADCAST	Nodes on which the radio receiver remains enabled when the node is idle (e.g. sleeping)
ZPS_E_APL_AF_BROADCAST	Only the Coordinator and Routers

The required broadcast mode is specified in the structure tsZCL_Address (see Section 6.1.4).

Parent topic: General Enumerations

Attribute Types (teZCL_ZCLAttributeType) The following enumerations are used to represent the attribute types in the/ZCL clusters:

```
typedef enum
{
    /* Null */
    E_ZCL_NULL          = 0x00,
    /* General Data */
    E_ZCL_GINT8         = 0x08,          // General 8 bit - not specified if signed
    E_ZCL_GINT16,
    E_ZCL_GINT24,
    E_ZCL_GINT32,
    E_ZCL_GINT40,
    E_ZCL_GINT48,
    E_ZCL_GINT56,
    E_ZCL_GINT64,
    /* Logical */
    E_ZCL_BOOL          = 0x10,
    /* Bitmap */
    E_ZCL_BMAP8         = 0x18,          // 8 bit bitmap
    E_ZCL_BMAP16,
    E_ZCL_BMAP24,
    E_ZCL_BMAP32,
    E_ZCL_BMAP40,
    E_ZCL_BMAP48,
    E_ZCL_BMAP56,
    E_ZCL_BMAP64,
    /* Unsigned Integer */
    E_ZCL_UINT8         = 0x20,          // Unsigned 8 bit
    E_ZCL_UINT16,
    E_ZCL_UINT24,
    E_ZCL_UINT32,
    E_ZCL_UINT40,
    E_ZCL_UINT48,
    E_ZCL_UINT56,
    E_ZCL_UINT64,
    /* Signed Integer */
    E_ZCL_INT8          = 0x28,          // Signed 8 bit
    E_ZCL_INT16,
```

(continues on next page)

(continued from previous page)

```
    E_ZCL_INT24,
    E_ZCL_INT32,
    E_ZCL_INT40,
    E_ZCL_INT48,
    E_ZCL_INT56,
    E_ZCL_INT64,
    /* Enumeration */
    E_ZCL_ENUM8          = 0x30,          // 8 Bit enumeration
    E_ZCL_ENUM16,
    /* Floating Point */
    E_ZCL_FLOAT_SEMI     = 0x38,       // Semi precision
    E_ZCL_FLOAT_SINGLE,                // Single precision
    E_ZCL_FLOAT_DOUBLE,                // Double precision
    /* String */
    E_ZCL_OSTRING        = 0x41,       // Octet string
    E_ZCL_CSTRING,                // Character string
    E_ZCL_LOSTRING,                // Long octet string
    E_ZCL_LCSTRING,                // Long character string
    /* Ordered Sequence */
    E_ZCL_ARRAY          = 0x48,
    E_ZCL_STRUCT         = 0x4c,
    E_ZCL_SET            = 0x50,
    E_ZCL_BAG            = 0x51,
    /* Time */
    E_ZCL_TOD            = 0xe0,          // Time of day
    E_ZCL_DATE,                        // Date
    E_ZCL_UTCT,                        // UTC Time
    /* Identifier */
    E_ZCL_CLUSTER_ID     = 0xe8,          // Cluster ID
    E_ZCL_ATTRIBUTE_ID,                // Attribute ID
    E_ZCL_BACNET_OID,                  // BACnet OID
    /* Miscellaneous */
    E_ZCL_IEEE_ADDR      = 0xf0,       // 64 Bit IEEE Address
    E_ZCL_KEY_128,                // 128 Bit security key
    /* Unknown */
    E_ZCL_UNKNOWN        = 0xff
} teZCL_ZCLAttributeType;
```

Parent topic: General Enumerations

Command Status (teZCL_CommandStatus) The following enumerations are used to indicate the status of a command:

```
typedef enum
{
    E_ZCL_CMDS_SUCCESS =0x00,
    E_ZCL_CMDS_FAILURE,
    E_ZCL_CMDS_NOT_AUTHORIZED =0x7e,
    E_ZCL_CMDS_RESERVED_FIELD_NOT_ZERO,
    E_ZCL_CMDS_MALFORMED_COMMAND =0x80,
    E_ZCL_CMDS_UNSUP_CLUSTER_COMMAND,
    E_ZCL_CMDS_UNSUP_GENERAL_COMMAND,
    E_ZCL_CMDS_UNSUP_MANUF_CLUSTER_COMMAND,
    E_ZCL_CMDS_UNSUP_MANUF_GENERAL_COMMAND,
    E_ZCL_CMDS_INVALID_FIELD,
    E_ZCL_CMDS_UNSUPPORTED_ATTRIBUTE,
    E_ZCL_CMDS_INVALID_VALUE,
    E_ZCL_CMDS_READ_ONLY,
    E_ZCL_CMDS_INSUFFICIENT_SPACE,
```

(continues on next page)

(continued from previous page)

```
    E_ZCL_CMDS_DUPLICATE_EXISTS,
    E_ZCL_CMDS_NOT_FOUND,
    E_ZCL_CMDS_UNREPORTABLE_ATTRIBUTE,
    E_ZCL_CMDS_INVALID_DATA_TYPE,
    E_ZCL_CMDS_INVALID_SELECTOR,
    E_ZCL_CMDS_WRITE_ONLY,
    E_ZCL_CMDS_INCONSISTENT_STARTUP_STATE,
    E_ZCL_CMDS_DEFINED_OUT_OF_BAND,
    E_ZCL_CMDS_INCONSISTENT,
    E_ZCL_CMDS_ACTION_DENIED,
    E_ZCL_CMDS_TIMEOUT,
    E_ZCL_CMDS_HARDWARE_FAILURE =0xc0,
    E_ZCL_CMDS_SOFTWARE_FAILURE,
    E_ZCL_CMDS_CALIBRATION_ERROR,
    E_ZCL_CMDS_UNSUPPORTED_CLUSTER,
    E_ZCL_CMDS_ENUM_END
} teZCL_CommandStatus;
```

The above enumerations are described in the table below.

Enumeration	Description
E_ZCL_CMDS_SUCCESS	Command was successful
E_ZCL_CMDS_FAILURE	Command was unsuccessful
E_ZCL_CMDS_NOT_AUTHORIZED	Sender does not have authorisation to issue the command
E_ZCL_CMDS_RESERVED_FIELD_N	A reserved field of command is not set to zero
E_ZCL_CMDS_MALFORMED_COM	Command has missing fields or invalid field values
E_ZCL_CMDS_UNSUP_CLUSTER_C	The specified cluster has not been registered with the ZCL on the device
E_ZCL_CMDS_UNSUP_GENERAL_C	Command does not have a handler enabled in the **zcl_options.h** file
E_ZCL_CMDS_UNSUP_MANUF_CL	Manufacturer-specific cluster command is not sup-ported or has unknown manufacturer code
E_ZCL_CMDS_UNSUP_MANUF_GE	Manufacturer-specific ZCL command is not sup-ported or has unknown manufacturer code
E_ZCL_CMDS_INVALID_FIELD	Command has field which contains invalid value
E_ZCL_CMDS_UNSUPPORTED_ATT	Specified attribute is not supported on the device
E_ZCL_CMDS_INVALID_VALUE	Specified attribute value is out of range or a reserved value
E_ZCL_CMDS_READ_ONLY	Attempt to write to read-only attribute
E_ZCL_CMDS_INSUFFICIENT_SPA	Not enough memory space to perform requested operation
E_ZCL_CMDS_DUPLICATE_EXISTS	Attempt made to create a table entry that already exists in the target table
E_ZCL_CMDS_NOT_FOUND	Requested information cannot be found
E_ZCL_CMDS_UNREPORTABLE_AT	Periodic reports cannot be produced for this attribute
E_ZCL_CMDS_INVALID_DATA_TYP	Invalid data type specified for attribute
E_ZCL_CMDS_INVALID_SELECTOR	Incorrect selector for this attribute
E_ZCL_CMDS_WRITE_ONLY	Issuer of command does not have authorisation to read specified attribute
E_ZCL_CMDS_INCONSISTENT_STA	Setting the specified values would put device into an in-consistent state on start-up
E_ZCL_CMDS_DEFINED_OUT_OF_	Attempt has been made to write to attribute using an out-of-band method or not over-air
E_ZCL_CMDS_HARDWARE_FAILU	Command was unsuccessful due to hardware failure
E_ZCL_CMDS_SOFTWARE_FAILUR	Command was unsuccessful due to software failure
E_ZCL_CMDS_CALIBRATION_ERR	Error occurred during calibration
E_ZCL_CMDS_UNSUPPORTED_CL	The cluster is not supported

Parent topic: General Enumerations

Report Attribute Status (teZCL_ReportAttributeStatus) The following enumerations are used to indicate the status of a report attribute command.

```
typedef enum
{
    E_ZCL_ATTR_REPORT_OK = 0x00,
    E_ZCL_ATTR_REPORT_EP_MISMATCH,
    E_ZCL_ATTR_REPORT_ADDR_MISMATCH,
    E_ZCL_ATTR_REPORT_ERR
} teZCL_ReportAttributeStatus;
```

The above enumerations are described in the table below.

Enumeration	Description
E_ZCL_ATTR_REPORT_OK	Indicates that report is valid
E_ZCL_ATTR_REPORT_EP_MISMATC	Indicates that source endpoint does not match endpoint in mirror
E_ZCL_ATTR_REPORT_ADDR_MISM	Indicates that source address does not match address in mirror
E_ZCL_ATTR_REPORT_ERR	Indicates that there is an error in the report

Parent topic: General Enumerations

Security Level (teZCL_ZCLSendSecurity) The following enumerations are used to indicate the security level for transmissions:

```
typedef enum
{
    E_ZCL_SECURITY_NETWORK = 0x00,
    E_ZCL_SECURITY_APPLINK,
    E_ZCL_SECURITY_TEMP_APPLINK,
    E_ZCL_SECURITY_ENUM_END
} teZCL_ZCLSendSecurity;
```

The above enumerations are described in the table below.

Enumeration	Description
E_ZCL_SECUF	Network-level security, using a network key
E_ZCL_SECUF	Application-level security, using an application link key
E_ZCL_SECUF	Temporary application-level security. This option is for internal use only. This is used for situations in which an application link key is to be used temporarily. For example, it can be used for an individual communication.

Parent topic: General Enumerations

Parent topic:*Enumerations and Status Codes*

General Return codes (ZCL Status) The following ZCL status enumerations are returned by many API functions to indicate the outcome of the function call.

```
typedef enum
{
  // General
  E_ZCL_SUCCESS = 0x0,
  E_ZCL_FAIL,                  // 01
  E_ZCL_ERR_PARAMETER_NULL,    // 02
  E_ZCL_ERR_PARAMETER_RANGE,   // 03
  E_ZCL_ERR_HEAP_FAIL,         // 04
  // Specific ZCL status codes
  E_ZCL_ERR_EP_RANGE,          // 05
  E_ZCL_ERR_EP_UNKNOWN,        // 06
  E_ZCL_ERR_SECURITY_RANGE,    // 07
  E_ZCL_ERR_CLUSTER_0,         // 08
  E_ZCL_ERR_CLUSTER_NULL,      // 09
  E_ZCL_ERR_CLUSTER_NOT_FOUND, // 10
  E_ZCL_ERR_CLUSTER_ID_RANGE,  // 11
  E_ZCL_ERR_ATTRIBUTES_NULL,   // 12
  E_ZCL_ERR_ATTRIBUTES_0,      // 13
  E_ZCL_ERR_ATTRIBUTE_WO,      // 14
  E_ZCL_ERR_ATTRIBUTE_RO,      // 15
  E_ZCL_ERR_ATTRIBUTES_ACCESS, // 16
  E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED, // 17
  E_ZCL_ERR_ATTRIBUTE_NOT_FOUND,   // 18
  E_ZCL_ERR_CALLBACK_NULL,         // 19
  E_ZCL_ERR_ZBUFFER_FAIL,          // 20
  E_ZCL_ERR_ZTRANSMIT_FAIL,        // 21
  E_ZCL_ERR_CLIENT_SERVER_STATUS,  // 22
  E_ZCL_ERR_TIMER_RESOURCE,        // 23
  E_ZCL_ERR_ATTRIBUTE_IS_CLIENT,   // 24
  E_ZCL_ERR_ATTRIBUTE_IS_SERVER,   // 25
  E_ZCL_ERR_ATTRIBUTE_RANGE,       // 26
  E_ZCL_ERR_ATTRIBUTE_MISMATCH,    // 27
  E_ZCL_ERR_KEY_ESTABLISHMENT_MORE_THAN_ONE_CLUSTER, //28
  E_ZCL_ERR_INSUFFICIENT_SPACE,    // 29
  E_ZCL_ERR_NO_REPORTABLE_CHANGE,  // 30
  E_ZCL_ERR_NO_REPORT_ENTRIES,     // 31
  E_ZCL_ERR_ATTRIBUTE_NOT_REPORTABLE, //32
  E_ZCL_ERR_ATTRIBUTE_ID_ORDER,    // 33
  E_ZCL_ERR_MALFORMED_MESSAGE,     // 34
  E_ZCL_ERR_MANUFACTURER_SPECIFIC, // 35
  E_ZCL_ERR_PROFILE_ID,            // 36
  E_ZCL_ERR_INVALID_VALUE,         // 37
  E_ZCL_ERR_CERT_NOT_FOUND,        // 38
  E_ZCL_ERR_CUSTOM_DATA_NULL,      // 39
  E_ZCL_ERR_TIME_NOT_SYNCHRONISED, // 40
  E_ZCL_ERR_SIGNATURE_VERIFY_FAILED, //41
  E_ZCL_ERR_ZRECEIVE_FAIL,         // 42
  E_ZCL_ERR_KEY_ESTABLISHMENT_END_POINT_NOT_FOUND, // 43
  E_ZCL_ERR_KEY_ESTABLISHMENT_CLUSTER_ENTRY_NOT_FOUND, // 44
  E_ZCL_ERR_KEY_ESTABLISHMENT_CALLBACK_ERROR, // 45
  E_ZCL_ERR_SECURITY_INSUFFICIENT_FOR_CLUSTER, // 46
  E_ZCL_ERR_CUSTOM_COMMAND_HANDLER_NULL_OR_RETURNED_ERROR, // 47
  E_ZCL_ERR_INVALID_IMAGE_SIZE,        // 48
  E_ZCL_ERR_INVALID_IMAGE_VERSION,     // 49
  E_ZCL_READ_ATTR_REQ_NOT_FINISHED,    // 50
  E_ZCL_DENY_ATTRIBUTE_ACCESS,         // 51
  E_ZCL_ERR_SECURITY_FAIL,             // 52
  E_ZCL_ERR_CLUSTER_COMMAND_NOT_FOUND,
  E_ZCL_ERR_ENUM_END
} teZCL_Status;
```

Enumeration	Description
E_ZCL_SUCCESS	Function call was successful in
E_ZCL_FAIL	Function call failed in its purpo
E_ZCL_ERR_PARAMETER_NULL	Specified parameter pointer wa
E_ZCL_ERR_PARAMETER_RANGE	A parameter value was out-of-r
E_ZCL_ERR_HEAP_FAIL	ZCL heap is out-of-memory
E_ZCL_ERR_EP_RANGE	Specified endpoint number was
E_ZCL_ERR_EP_UNKNOWN	Specified endpoint has not been
E_ZCL_ERR_SECURITY_RANGE	Security value is out-of-range
E_ZCL_ERR_CLUSTER_0	Specified endpoint has no cluste
E_ZCL_ERR_CLUSTER_NULL	Specified pointer to a cluster wa
E_ZCL_ERR_CLUSTER_NOT_FOUND	Specified cluster has not been r
E_ZCL_ERR_CLUSTER_ID_RANGE	Specified cluster ID was out-of-r
E_ZCL_ERR_ATTRIBUTES_NULL	Specified pointer to an attribute
E_ZCL_ERR_ATTRIBUTES_0	List of attributes to be read was
E_ZCL_ERR_ATTRIBUTE_WO	Attempt was made to read write
E_ZCL_ERR_ATTRIBUTE_RO	Attempt was made to write to r
E_ZCL_ERR_ATTRIBUTES_ACCESS	Error occurred while accessing
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED	Specified attribute was of unsu
E_ZCL_ERR_ATTRIBUTE_NOT_FOUND	Specified attribute was not four
E_ZCL_ERR_CALLBACK_NULL	Specified pointer to a callback f
E_ZCL_ERR_ZBUFFER_FAIL	No buffer available to transmit
E_ZCL_ERR_ZTRANSMIT_FAIL\ (See Note 1 below this table)	ZigBee PRO stack has reported a
E_ZCL_ERR_CLIENT_SERVER_STATUS	Cluster instance of wrong kind
E_ZCL_ERR_TIMER_RESOURCE	No timer resource was availabl
E_ZCL_ERR_ATTRIBUTE_IS_CLIENT	Attempt made by a cluster clien
E_ZCL_ERR_ATTRIBUTE_IS_SERVER	Attempt made by a cluster serv
E_ZCL_ERR_ATTRIBUTE_RANGE	Attribute value is out-of-range
E_ZCL_ERR_ATTRIBUTE_MISMATCH	Reserved for future use
E_ZCL_ERR_KEY_ESTABLISHMENT_MORE_THAN_ONE_CLUSTER	Attempt made to register more
E_ZCL_ERR_INSUFFICIENT_SPACE	Cluster does not have enough s
E_ZCL_ERR_NO_REPORTABLE_CHANGE	Reserved for future use
E_ZCL_ERR_NO_REPORT_ENTRIES	Reserved for future use
E_ZCL_ERR_ATTRIBUTE_NOT_REPORTABLE	Reserved for future use
E_ZCL_ERR_ATTRIBUTE_ID_ORDER (See note 2 below this table)	Attempt made to register a clus
E_ZCL_ERR_MALFORMED_MESSAGE	Received ZCL message is not fo
E_ZCL_ERR_ATTRIBUTE_ID_ORDER ** (See note 2 below this table)	Attempt made to register a clus
E_ZCL_ERR_MANUFACTURER_SPECIFIC **	Inconsistency in a manufacture
E_ZCL_ERR_ATTRIBUTE_ID_ORDER (See note 2 below this table)	Attempt made to register a clus
E_ZCL_ERR_PROFILE_ID (See Note 2 below this table)	Profile ID of a cluster is not vali
E_ZCL_ERR_INVALID_VALUE	An invalid value has been detec
E_ZCL_ERR_CERT_NOT_FOUND	Reserved for future use
E_ZCL_ERR_CUSTOM_DATA_NULL	Custom data associated with clu
E_ZCL_ERR_TIME_NOT_SYNCHRONISED	Time has not been synchronize
E_ZCL_ERR_SIGNATURE_VERIFY_FAILED	Reserved for future use
E_ZCL_ERR_ZRECEIVE_FAIL * (See Note 1 below this table)	ZigBee PRO stack has reported a
E_ZCL_ERR_KEY_ESTABLISHMENT_END_POINT_NOT_FOUND	Key Establishment endpoint has
E_ZCL_ERR_KEY_ESTABLISHMENT_CLUSTER_ENTRY_NOT_FOUND	Key Establishment cluster has r
E_ZCL_ERR_KEY_ESTABLISHMENT_CALLBACK_ERROR	Key Establishment cluster callb
E_ZCL_ERR_SECURITY_INSUFFICIENT_FOR_CLUSTER	Cluster that requires applicatio
E_ZCL_ERR_CUSTOM_COMMAND_HANDLER_NULL_OR_RETURNED_ERROR	No custom handler has been re
E_ZCL_ERR_INVALID_IMAGE_SIZE	OTA image size is not in the cor
E_ZCL_ERR_INVALID_IMAGE_VERSION	OTA image version is not in the
E_ZCL_READ_ATTR_REQ_NOT_FINISHED	'Read attributes' request not co
E_ZCL_DENY_ATTRIBUTE_ACCESS	Write access to attribute is deni
E_ZCL_ERR_SECURITY_FAIL	Security failure

Enumeration	Description
E_ZCL_ERR_CLUSTER_COMMAND_NOT_FOUND	The cluster command was not f
E_ZCL_ERR_INVALID_VALUE	Reserved for future use

Note1: ZigBee PRO stack raises an error which can be retrieved using **eZCL_GetLastZpsError()**.

Note2: This error code is returned by **eZCL_Register()**, used in designing custom clusters.

Parent topic:*Enumerations and Status Codes*

ZCL Event Enumerations The ZCL event types are enumerated in the teZCL_CallBackEventType structure below and described in *Table 22*. An event must be wrapped in a structure of type tsZCL_CallBackEvent, detailed in Section 6.2, with the eEventType field set to one of the enumerations in the table. The event must be passed into the ZCL using the function **vZCL_EventHandler()**, detailed in Section 5.1. Event handling is fully described in *Chapter 3*.

```
typedef enum
{
  E_ZCL_CBET_LOCK_MUTEX = 0x0,
  E_ZCL_CBET_UNLOCK_MUTEX,
  E_ZCL_CBET_UNHANDLED_EVENT,
  E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE,
  E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE,
  E_ZCL_CBET_READ_REQUEST,
  E_ZCL_CBET_REPORT_REQUEST,
  E_ZCL_CBET_DEFAULT_RESPONSE,
  E_ZCL_CBET_ERROR,
  E_ZCL_CBET_TIMER,
  E_ZCL_CBET_ZIGBEE_EVENT,
  E_ZCL_CBET_CLUSTER_CUSTOM,
  E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE,
  E_ZCL_CBET_WRITE_ATTRIBUTES,
  E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE,
  E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE,
  E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE,
  E_ZCL_CBET_REPORT_TIMEOUT,
  E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTE,
  E_ZCL_CBET_REPORT_ATTRIBUTES,
  E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE_RESPONSE,
  E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE,
  E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE,
  E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE_RESPONSE,
  E_ZCL_CBET_REPORT_READ_INDIVIDUAL_ATTRIBUTE_CONFIGURATION_RESPONSE,
  E_ZCL_CBET_REPORT_READ_ATTRIBUTE_CONFIGURATION_RESPONSE,
  E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_RESPONSE,
  E_ZCL_CBET_DISCOVER_ATTRIBUTES_RESPONSE,
  E_ZCL_CBET_CLUSTER_UPDATE,
  E_ZCL_CBET_ATTRIBUTE_REPORT_MIRROR,
  E_ZCL_CBET_REPORT_REQUEST,
  E_ZCL_CBET_ENABLE_MS_TIMER,
  E_ZCL_CBET_DISABLE_MS_TIMER,
  E_ZCL_CBET_TIMER_MS,
  E_ZCL_CBET_ZGP_DATA_IND_ERROR,
  E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_RECEIVED_RESPONSE,
  E_ZCL_CBET_DISCOVER_COMMAND_RECEIVED_RESPONSE,
  E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_GENERATED_RESPONSE,
  E_ZCL_CBET_DISCOVER_COMMAND_GENERATED_RESPONSE,
  E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_EXTENDED_RESPONSE,
```

(continues on next page)

(continued from previous page)

```
  E_ZCL_CBET_DISCOVER_ATTRIBUTES_EXTENDED_RESPONSE,
  E_ZCL_CBET_ENUM_END
} teZCL_CallBackEventType;
```

The above enumerations are described in the table below.

Event Type Enumeration	Description
E_ZCL_CBET_L	Indicates that a mutex needs to be locked by the application. This event can be generated only when cooperative tasks are disabled in the compile-time options

(see Section 1.3)

| |E_ZCL_CBET_UNLOCK_MUTEX|Indicates that a mutex needs to be unlocked by the application. This event can be generated only when cooperative tasks are disabled in the compile-time options (see Section 1.3)| |E_ZCL_CBET_UNHANDLED_EVENT|Indicates that a stack event has been received that cannot be handled by the ZCL (for example, a Data Confirm)| |E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE|Generated for each attribute included in a 'read attributes' response| |E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE|Indicates that a 'read attributes' response has been received| |E_ZCL_CBET_READ_REQUEST|Indicates that a 'read attributes' request has been received (giving an opportunity for the local application to update the shared structure before it is read)| |E_ZCL_CBET_DEFAULT_RESPONSE|Indicates that a ZCL default response message has been received (which indicates an error or that a command has been processed)| |E_ZCL_CBET_ERROR|Indicates that a stack event has been received that can-not be handled by the ZCL| |E_ZCL_CBET_TIMER|Indicates that a one-second tick of the real-time clock has occurred or that the ZCL timer has expired| |E_ZCL_CBET_ZIGBEE_EVENT|Indicates that a ZigBee PRO stack event has occurred| |E_ZCL_CBET_CLUSTER_CUSTOM|Indicates that a custom event which is specific to a cluster has occurred| |E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE|Indicates that an attempt has been made to write an attribute in the shared structure, following a 'write attributes' request, and indicates success or failure| |E_ZCL_CBET_WRITE_ATTRIBUTES|Indicates that all the relevant attributes have been written in the shared structure, following a 'write attributes' request| |E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE|Generated for each attribute included in a 'write attributes' response (this event contains only those attributes for which the writes have failed)| |E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE|Indicates that a 'write attributes' response has been received and has been parsed| |E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE|Generated for each attribute included in a received 'write attributes' request, and prompts the application to perform a range check on the new attribute value and to decide whether a write access to the relevant attribute in the shared structure are allowed or disallowed| |E_ZCL_CBET_REPORT_TIMEOUT|Indicates that an attribute report is overdue| |E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTE|Generated for each attribute included in a received attribute report| |E_ZCL_CBET_REPORT_ATTRIBUTES|Indicates that all attributes included in a received attribute report have been parsed| |E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE_RESPONSE|Generated for each attribute included in a 'configure attributes' response| |E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE|Indicates that all attributes included in a 'configure reporting' request have been parsed| |E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE|Generated for each attribute included in a 'configure reporting' request| |E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE_RESPONSE|Indicates that all attributes included in a 'configure reporting' response have been reported| |E_ZCL_CBET_REPORT_READ_INDIVIDUAL

_ATTRIBUTE_CONFIGURATION_RESPONSE

|Generated for each attribute included in a 'read reporting configuration' response| |E_ZCL_CBET_REPORT_READ_ATTRIBUTE_CONFIGURATION_RESPONSE|Indicates that all attributes included in a 'read reporting configuration' response have been reported| |E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_RESPONSE|Generated for each attribute included in a 'discover attributes' response| |E_ZCL_CBET_DISCOVER_ATTRIBUTES

_RESPONSE

|Indicates that all attributes included in a 'discover attributes' response have been reported| |E_ZCL_CBET_CLUSTER_UPDATE|Indicates that a cluster attribute value may have been changed on the local device| |E_ZCL_CBET_ENABLE_MS_TIMER|Indicates that a millisecond timer needs to be started| |E_ZCL_CBET_DISABLE_MS_TIMER|Indicates that a millisecond timer needs to be stopped| |E_ZCL_CBET_TIMER_MS|Indicates that a millisecond timer has expired| |E_ZCL_CBET_ZGP_DATA_IND_ERROR|Indicates that a ZigBee Green Power data indication error has occurred| |E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_RECEIVED_RESPONSE|Generated for each command (that can be received) included in a 'command discovery' response| |E_ZCL_CBET_DISCOVER_COMMAND_RECEIVED_RESPONSE|Indicates that all commands (that can be received) included in a 'command discovery' response have been reported| |E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_GENERATED_RESPONSE|Generated for each command (that can be generated) included in a 'command discovery' response| |E_ZCL_CBET_DISCOVER_COMMAND_GENERATED_RESPONSE|Indicates that all commands (that can be generated) included in a 'command discovery' response have been reported| |E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_EXTENDED_RESPONSE|Generated for each attribute included in a 'discover attributes extended' response| |E_ZCL_CBET_DISCOVER_ATTRIBUTES_EXTENDED_RESPONSE|Indicates that all attributes included in a 'discover attributes extended' response have been reported| |E_ZCL_CBET_REPORT_TIMEOUT|Reserved for future use| |E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTE|Reserved for future use| |E_ZCL_CBET_REPORT_ATTRIBUTES|Reserved for future use| |E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE_RESPONSE|Reserved for future use| |E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE|Reserved for future use| |E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE|Reserved for future use| |E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE_RESPONSE|Reserved for future use| |E_ZCL_CBET_REPORT_READ_INDIVIDUAL_ATTRIBUTE_CONFIGURATION_RESPONSE|Reserved for future use| |E_ZCL_CBET_REPORT_READ_ATTRIBUTE_CONFIGURATION_RESPONSE|Reserved for future use| |E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_RESPONSE|Reserved for future use| |E_ZCL_CBET_DISCOVER_ATTRIBUTES_RESPONSE|Reserved for future use|

Note: The structure teZCL_CallBackEventType is extended by the EZ-mode Commissioning module with the events listed and described in *Section 40.5*. These events are only included if this module is used, in which case they are added after E_ZCL_CBET_ENUM_END.

Parent topic:*Enumerations and Status Codes*

Part III: General Clusters This Part, General Clusters, comprises fifteen chapters:

- *Chapter 8* details the **Basic** cluster
- *Chapter 9* details the **Power Configuration** cluster
- *Chapter 10* details the **Device Temperature Configuration** cluster
- *Chapter 11* details the **Identify** cluster
- *Chapter 12* details the **Groups** cluster
- *Chapter 13* details the **Scenes** cluster
- *Chapter 14* details the **On/Off** cluster
- *Chapter 15* details the **On/Off Switch Configuration** cluster
- *Chapter 16* details the **Level Control** cluster

- *Chapter 17* details the **Alarms** cluster
- *Chapter 18* details the **Time** cluster, as well as the use of ZCL time
- Chapter 19 details the **Input and Output** clusters
- *Chapter 20* details the **Poll Control** cluster
- *Chapter 21* details the **Power Profile** cluster
- *Chapter 22* details the **Diagnostics** cluster

Basic Cluster This chapter details the Basic cluster which is a mandatory cluster for all ZigBee devices.

The Basic cluster has a Cluster ID of 0x0000.

Overview All devices implement the Basic cluster as a Server-side (input) cluster, so the cluster is able to store attributes and respond to commands relating to these attributes. The cluster's attributes hold basic information about the node (and apply to devices associated with all active endpoints on the host node). The information that can potentially be stored in this cluster comprises: ZCL version, application version, stack version, hardware version, manufacturer name, model identifier, date, power source.

Note: The Basic cluster can also be implemented as a Client-side (output) cluster to allow the host device to act as a commissioning tool.

The Basic cluster contains only two mandatory attributes, the remaining attributes being optional - see Section 8.2.

Note: Since the Basic cluster contains information about the entire node, only one set of Basic cluster attributes must be stored on the node, even if there are multiple instances of the Basic cluster server across multiple devices/endpoints. All cluster instances must refer to the same structure containing the attribute values.

The Basic cluster is enabled by defining CLD_BASIC in the **zcl_options.h** file.

A Basic cluster instance can act as a client and/or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Basic cluster are fully detailed in Section 8.6.

Parent topic:*Basic Cluster*

Basic Cluster structure and attributes The Basic cluster is contained in the following tsCLD_Basic structure:

```
typedef struct
{
#ifdef BASIC_SERVER
    zuint8                  u8ZCLVersion;
#ifdef CLD_BAS_ATTR_APPLICATION_VERSION
    zuint8                  u8ApplicationVersion;
#endif
#ifdef CLD_BAS_ATTR_STACK_VERSION
    zuint8                  u8StackVersion;
#endif
#ifdef CLD_BAS_ATTR_HARDWARE_VERSION
    zuint8                  u8HardwareVersion;
#endif
```

(continues on next page)

(continued from previous page)

```
#ifdef CLD_BAS_ATTR_MANUFACTURER_NAME
    tsZCL_CharacterString sManufacturerName;
uint8                 au8ManufacturerName[32];
#endif
#ifdef CLD_BAS_ATTR_MODEL_IDENTIFIER
    tsZCL_CharacterString sModelIdentifier;
    uint8                 au8ModelIdentifier[32];
#endif
#ifdef CLD_BAS_ATTR_DATE_CODE
    tsZCL_CharacterString sDateCode;
    uint8                 au8DateCode[16];
#endif
    zenum8                ePowerSource;
#ifdef CLD_BAS_ATTR_GENERIC_DEVICE_CLASS
    zenum8                eGenericDeviceClass;
#endif
#ifdef CLD_BAS_ATTR_GENERIC_DEVICE_TYPE
    zenum8                eGenericDeviceType;
#endif
#ifdef CLD_BAS_ATTR_PRODUCT_CODE
    tsZCL_OctetString     sProductCode;
uint8
au8ProductCode[CLD_BASIC_MAX_NUMBER_OF_BYTES_PRODUCT_CODE];
#endif
#ifdef CLD_BAS_ATTR_PRODUCT_URL
    tsZCL_CharacterString  sProductURL;
    uint8
    au8ProductURL[CLD_BASIC_MAX_NUMBER_OF_BYTES_PRODUCT_URL];
#endif
#ifdef CLD_BAS_ATTR_LOCATION_DESCRIPTION
    tsZCL_CharacterString  sLocationDescription;
    uint8                  au8LocationDescription[16];
#endif
#ifdef CLD_BAS_ATTR_PHYSICAL_ENVIRONMENT
    zenum8                 u8PhysicalEnvironment;
#endif
#ifdef CLD_BAS_ATTR_DEVICE_ENABLED
    zbool                  bDeviceEnabled;
#endif
#ifdef CLD_BAS_ATTR_ALARM_MASK
    zbmap8                 u8AlarmMask;
#endif
#ifdef CLD_BAS_ATTR_DISABLE_LOCAL_CONFIG
    zbmap8                 u8DisableLocalConfig;
#endif
#ifdef CLD_BAS_ATTR_SW_BUILD_ID
    tsZCL_CharacterString sSWBuildID;
    uint8                 au8SWBuildID[16];
#endif
#endif
    zuint16               u16ClusterRevision;
} tsCLD_Basic;
```

where:

- u8ZCLVersion is an 8-bit version number which represents a published set of foundation items, such as global commands and functional descriptions. Currently this should be set to 2.
- u8ApplicationVersion is an optional 8-bit attribute which represents the version of the application (and is manufacturer-specific)
- u8StackVersion is an optional 8-bit attribute which represents the version of the ZigBee stack

used (and is manufacturer-specific)

- u8HardwareVersion is an optional 8-bit attribute which represents the version of the hardware used for the device (and is manufacturer-specific)
- The following optional pair of attributes are used to store the name of the manufacturer of the device:
  - sManufacturerName is a tsZCL_CharacterString structure (see Section 6.1.14) for a string of up to 32 characters representing the manufacturer's name
  - au8ManufacturerName[32] is a byte-array which contains the character data bytes representing the manufacturer's name
- The following optional pair of attributes are used to store the identifier for the model of the device:
  - sModelIdentifier is a tsZCL_CharacterString structure (see Section 6.1.14) for a string of up to 32 characters representing the model identifier
  - au8ModelIdentifier[32] is a byte-array which contains the character data bytes representing the model identifier
- The following optional pair of attributes are used to store manufacturing information about the device:
  - sDateCode is a tsZCL_CharacterString structure (see Section6.1.14) for a string of up to 16 characters in which the 8 most significant characters contain the date of manufacture in the format YYYYMMDD and the 8 least significant characters contain manufacturer-defined information such as country of manufacture, factory identifier, production line identifier
  - au8DateCode[16] is a byte-array which contains the character data bytes representing the manufacturing information

Note: The application device code automatically sets two of the fields of sDataCode. The field sDataCode.pu8Data is set to point at au8DateCodeand the field sDataCode.u8MaxLengthis set to 16 (see Section 6.1.14 for details of these fields).

- ePowerSource is an 8-bit value in which seven bits indicate the primary power source for the device (e.g. battery) and one bit indicates whether there is a secondary power source for the device. Enumerations are provided to cover all possibilities - see Section 8.5.2

Note: The power source in the Basic cluster is completely unrelated to the Node Power descriptor in the ZigBee PRO stack. The power source in the ZigBee PRO stack is set using the ZPS Configuration Editor.

- eGenericDeviceClass is an optional attribute that identifies the field of application in which the local device type operates (see eGenericDeviceType below). Enumerations are provided - see Section 8.5.3. Currently, the attribute is used only in lighting applications, for which the value is 0x00 (all other values are reserved).
- eGenericDeviceType is an optional attribute that identifies the local device type. Enumerations are provided to cover the different possibilities - see Section 8.5.4. Currently, the attribute is used only in lighting applications.
- The following optional pair of attributes are used to store a code for the product (this attribute may be used in lighting applications only):
  - sProductCode is a tsZCL_OctetString structure (see Section 6.1.14) for a string representing the product code - the maximum number of characters is defined at compile-time (see Section 8.6) using the macro CLD_BASIC_MAX_NUMBER_OF_BYTES_PRODUCT_CODE.
  - au8ProductCode[``] is a byte-array which contains the character data bytes representing the product code - the number of array elements, and therefore characters, is determined at compile-time, as indicated above.

- The following optional pair of attributes are used to store a URL for the product (this attribute may be used in lighting application only):
  - sProductURL is a tsZCL_CharacterString structure (see Section 6.1.14) for a character string representing the product URL - the maximum number of characters is defined at compile-time (see Section 8.6) using the macro CLD_BASIC_MAX_NUMBER_OF_BYTES_PRODUCT_URL.
  - au8ProductURL[``] is a byte-array which contains the character data bytes representing the product URL - the number of array elements, and therefore characters, is determined at compile-time, as indicated above.
- The following optional pair of attributes relates to the location of the device:
  - sLocationDescription is a tsZCL_CharacterString structure (see Section 6.1.14) for a string of up to 16 characters representing the location of the device
  - au8LocationDescription[16] is a byte-array which contains the character data bytes representing the location of the device
- u8PhysicalEnvironment is an optional 8-bit attribute which indicates the physical environment of the device. Enumerations are provided to cover the different possibilities - see Section 8.5.5.
- bDeviceEnabled is an optional Boolean attribute which indicates whether the device is enabled (TRUE) or disabled (FALSE). A disabled device cannot send or respond to application level commands other than commands to read or write attributes
- u8AlarmMask is an optional bitmap indicating the general alarms that can be generated (Bit 0 - general software alarm, Bit 1 - general hardware alarm)
- u8DisableLocalConfig is an optional bitmap allowing the local user interface of the device to be disabled (Bit 0 - 'Reset to factory defaults' buttons, Bit 1 - 'Device configuration' buttons)
- The following optional pair of attributes are used to store a manufacturer-specific software build identifier:
  - sSWBuildID is a tsZCL_CharacterString structure (see Section 6.1.14) for a string of up to 16 characters representing the software build identifier
  - au8SWBuildID[16] is a byte-array which contains the character data bytes representing the software build identifier.
  - u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4

The Basic cluster structure contains three mandatory elements: u``8ZCLVersion, e``PowerSource and u16ClusterRevision. The remaining elements are optional, each being enabled/disabled through a corresponding macro defined in the **zcl_options.h** file - for example, the attribute u8ApplicationVersion is enabled/disabled using the enumeration CLD_BAS_ATTR_APPLICATION_VERSION (see Section 8.3).

The mandatory attribute settings are described further in Section 8.3.

Parent topic:*Basic Cluster*

Mandatory Attribute Settings The application must set the values of the mandatory u8ZCLVersion, ePowerSource and u16ClusterRevision fields of the Basic cluster structure so that other devices can read them. This should be done immediately after calling the endpoint registration function for the device - for example, **eZLO_RegisterDimmableLightEndPoint()**.

These values can be set by calling the **eZCL_WriteLocalAttributeValue()** function with the appropriate input values. Alternatively, they can be set by writing to the relevant members of the

shared structure of the device, as illustrated below, where sLight or sSwitch is the device that is registered using the registration function.

On a Dimmable Light:

```
sLight.sBasicCluster.u8ZCLVersion = 0x01;
sLight.sBasicCluster.ePowerSource = E_CLD_BAS_PS_SINGLE_PHASE_MAINS;
sLight.sBasicClusteru.16ClusterRevision = CLD_BAS_CLUSTER_REVISION;
```

On a battery-powered Dimmer Switch:

```
sSwitch.sLocalBasicCluster.u8ZCLVersion = 0x01;
sSwitch.sLocalBasicCluster.ePowerSource = E_CLD_BAS_PS_BATTERY;
sLight.sBasicClusteru.16ClusterRevision = CLD_BAS_CLUSTER_REVISION;
```

Parent topic:*Basic Cluster*

Functions The following two Basic cluster functions are provided in the NXP implementation of the ZCL:

- eCLD_BasicCreateBasic
- eCLD_BasicCommandResetToFactoryDefaultsSend

eCLD_BasicCreateBasic

```
teZCL_Status eCLD_BasicCreateBasic(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Basic cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Basic cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device (for example, a Simple Sensor) is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be the first Basic cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Basic cluster. The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* : Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Basic cluster. This parameter can refer to a pre-filled structure called sCLD_Basic which is provided in the **Basic.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_Basic which defines the attributes of Basic cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Functions

eCLD_BasicCommandResetToFactoryDefaultsSend

```
teZCL_Status eCLD_BasicCommandResetToFactoryDefaultsSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be called on a client device to send a 'Reset To Factory Defaults' command, requesting the recipient server device to reset to its factory defaults. The recipient device generates a callback event on the endpoint on which the Basic cluster was registered.

If used, the 'Reset To Factory Defaults' command must be enabled in the compile-time options on both the client and server, as described in Section 8.6.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

Parent topic:*Basic Cluster*

Enumerations

teCLD_BAS_ClusterID The following structure contains the enumerations used to identify the attributes of the Basic cluster.

```
typedef enum
{
    E_CLD_BAS_ATTR_ID_ZCL_VERSION = 0x0000, /* Mandatory */
    E_CLD_BAS_ATTR_ID_APPLICATION_VERSION,
    E_CLD_BAS_ATTR_ID_STACK_VERSION,
    E_CLD_BAS_ATTR_ID_HARDWARE_VERSION,
    E_CLD_BAS_ATTR_ID_MANUFACTURER_NAME,
    E_CLD_BAS_ATTR_ID_MODEL_IDENTIFIER,
    E_CLD_BAS_ATTR_ID_DATE_CODE,
    E_CLD_BAS_ATTR_ID_POWER_SOURCE, /* Mandatory */
    E_CLD_BAS_ATTR_ID_LOCATION_DESCRIPTION = 0x0010,
    E_CLD_BAS_ATTR_ID_PHYSICAL_ENVIRONMENT,
    E_CLD_BAS_ATTR_ID_DEVICE_ENABLED,
    E_CLD_BAS_ATTR_ID_ALARM_MASK,
    E_CLD_BAS_ATTR_ID_DISABLE_LOCAL_CONFIG,
    E_CLD_BAS_ATTR_ID_SW_BUILD_ID = 0x4000
} teCLD_BAS_ClusterID;
```

Parent topic: Enumerations

teCLD_BAS_PowerSource The following enumerations are used in the Basic cluster to specify the power source for a device (see above):

```
typedef enum
{
    E_CLD_BAS_PS_UNKNOWN = 0x00,
    E_CLD_BAS_PS_SINGLE_PHASE_MAINS,
    E_CLD_BAS_PS_THREE_PHASE_MAINS,
    E_CLD_BAS_PS_BATTERY,
    E_CLD_BAS_PS_DC_SOURCE,
    E_CLD_BAS_PS_EMERGENCY_MAINS_CONSTANTLY_POWERED,
    E_CLD_BAS_PS_EMERGENCY_MAINS_AND_TRANSFER_SWITCH,
    E_CLD_BAS_PS_UNKNOWN_BATTERY_BACKED = 0x80,
    E_CLD_BAS_PS_SINGLE_PHASE_MAINS_BATTERY_BACKED,
    E_CLD_BAS_PS_THREE_PHASE_MAINS_BATTERY_BACKED,
    E_CLD_BAS_PS_BATTERY_BATTERY_BACKED,
    E_CLD_BAS_PS_DC_SOURCE_BATTERY_BACKED,
    E_CLD_BAS_PS_EMERGENCY_MAINS_CONSTANTLY_POWERED_BATTERY_BACKED,
    E_CLD_BAS_PS_EMERGENCY_MAINS_AND_TRANSFER_SWITCH_BATTERY_BACKED,
} teCLD_BAS_PowerSource;
```

The power source enumerations are described in the table below.

Enumeration	Description
E_CLD_BAS_PS_UNKNOWN	Unknown power source
E_CLD_BAS_PS_SINGLE_PHASE_MAINS	Single-phase mains powered
E_CLD_BAS_PS_THREE_PHASE_MAINS	Three-phase mains powered
E_CLD_BAS_PS_BATTERY	Battery powered
E_CLD_BAS_PS_DC_SOURCE	DC source
E_CLD_BAS_PS_EMERGENCY_MAINS_CONSTAN	Constantly powered from emergency mains supply
E_CLD_BAS_PS_EMERGENCY_MAINS_AND_TRA	Powered from emergency mains supply via transfer switch
E_CLD_BAS_PS_UNKNOWN_BATTERY_BACKED	Unknown power source but battery back-up
E_CLD_BAS_PS_SINGLE_PHASE_MAINS_BATTE	Single-phase mains powered with battery back-up
E_CLD_BAS_PS_THREE_PHASE_MAINS_BATTEF	Three-phase mains powered with battery back-up
E_CLD_BAS_PS_BATTERY_BATTERY_BACKED	Battery powered with battery back-up
E_CLD_BAS_PS_DC_SOURCE_BATTERY_BACKED	DC source with battery back-up
E_CLD_BAS_PS_EMERGENCY_MAINS_CONSTAN	Constantly powered from emergency mains supply with battery back-up
E_CLD_BAS_PS_EMERGENCY_MAINS_AND_TRA	Powered from emergency mains supply via transfer switch with battery back-up

Parent topic: Enumerations

teCLD_BAS_GenericDeviceClass The following enumerations are used in the Basic cluster to specify the Device Classes:

```
typedef enum
{
    E_CLD_BAS_GENERIC_DEVICE_CLASS_LIGHTING        = 0x00,
} teCLD_BAS_GenericDeviceClass;
```

Parent topic: Enumerations

eCLD_BAS_GenericDeviceType The following enumerations are used in the Basic cluster to specify the Device Types:

```
typedef enum
{
    E_CLD_BAS_GENERIC_DEVICE_TYPE_INCANDESCENT        = 0x00,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_SPOTLIGHT_HALOGEN,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_HALOGEN_BULB,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_CFL,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_LINEAR_FLUORESCENT,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_LED_BULB,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_SPOTLIGHT_LED,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_LED_STRIP,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_LED_TUBE,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_INDOOR_LUMINAIRE,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_OUTDOOR_LUMINAIRE,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_PENDANT_LUMINAIRE,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_FLOOR_STANDING_LUMINAIRE,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_CONTROLLER          = 0xE0,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_WALL_SWITCH,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_PORTABLE_REMOTE_CONTROLLER,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_MOTION_OR_LIGHT_SENSOR,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_ACTUATOR            = 0xF0,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_WALL_SOCKET,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_GATEWAY_OR_BRIDGE,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_PLUG_IN_UNIT,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_RETROFIT_ACTUATOR,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_UNSPECIFIED         = 0xFF
} teCLD_BAS_GenericDeviceType;
```

Parent topic: Enumerations

teCLD_BAS_PhysicalEnvironment The following enumerations are used in the Basic cluster to specify the Physical Environment:

```
typedef enum
{
    E_CLD_BAS_PE_UNSPECIFIED              = 0x00,
    E_CLD_BAS_PE_MIRROR,
    E_CLD_BAS_PE_ATRIUM,
    E_CLD_BAS_PE_BAR,
    E_CLD_BAS_PE_COURTYARD,
    E_CLD_BAS_PE_BATHROOM,
    E_CLD_BAS_PE_BEDROOM,
    E_CLD_BAS_PE_BILLIARD_ROOM,
    E_CLD_BAS_PE_UTILITY_ROOM,
    E_CLD_BAS_PE_CELLAR,
    E_CLD_BAS_PE_STORAGE_CLOSET,
    E_CLD_BAS_PE_THREATER,
    E_CLD_BAS_PE_OFFICE_0x0B,
    E_CLD_BAS_PE_DECK,
    E_CLD_BAS_PE_DEN,
    E_CLD_BAS_PE_DINNING_ROOM,
    E_CLD_BAS_PE_ELECTRICAL_ROOM,
    E_CLD_BAS_PE_ELEVATOR,
    E_CLD_BAS_PE_ENTRY,
    E_CLD_BAS_PE_FAMILY_ROOM,
    E_CLD_BAS_PE_MAIN_FLOOR,
    E_CLD_BAS_PE_UPSTAIRS,
    E_CLD_BAS_PE_DOWNSTAIRS,
```

(continues on next page)

(continued from previous page)

```
E_CLD_BAS_PE_BASEMENT_LOWER_LEVEL,
E_CLD_BAS_PE_GALLERY,
E_CLD_BAS_PE_GAME_ROOM,
E_CLD_BAS_PE_GARAGE,
E_CLD_BAS_PE_GYM,
E_CLD_BAS_PE_HALLWAY,
E_CLD_BAS_PE_HOUSE,
E_CLD_BAS_PE_KITCHEN,
E_CLD_BAS_PE_LAUNDRY_ROOM,
E_CLD_BAS_PE_LIBRARY,
E_CLD_BAS_PE_MASTER_BEDROOM,
E_CLD_BAS_PE_MUD_ROOM,
E_CLD_BAS_PE_NURSERY,
E_CLD_BAS_PE_PANTRY,
E_CLD_BAS_PE_OFFICE_0X24,
E_CLD_BAS_PE_OUTSIDE,
E_CLD_BAS_PE_POOL,
E_CLD_BAS_PE_PORCH,
E_CLD_BAS_PE_SEWING_ROOM,
E_CLD_BAS_PE_SITTING_ROOM,
E_CLD_BAS_PE_STAIRWAY,
E_CLD_BAS_PE_YARD,
E_CLD_BAS_PE_ATTIC,
E_CLD_BAS_PE_HOT_TUB,
E_CLD_BAS_PE_LIVING_ROOM_0X2E,
E_CLD_BAS_PE_SAUNA,
E_CLD_BAS_PE_SHOP_WORKSHOP,
E_CLD_BAS_PE_GUEST_BEDROOM,
E_CLD_BAS_PE_GUEST_BATH,
E_CLD_BAS_PE_POWDER_ROOM,
E_CLD_BAS_PE_BACK_YARD,
E_CLD_BAS_PE_FRONT_YARD,
E_CLD_BAS_PE_PATIO,
E_CLD_BAS_PE_DRIVEWAY,
E_CLD_BAS_PE_SUN_ROOM,
E_CLD_BAS_PE_LIVING_ROOM_0X39,
E_CLD_BAS_PE_SPA,
E_CLD_BAS_PE_WHIRLPOOL,
E_CLD_BAS_PE_SHED,
E_CLD_BAS_PE_EQUIPMENT_STORAGE,
E_CLD_BAS_PE_HOBBY_CRAFT_ROOM,
E_CLD_BAS_PE_FOUNTAIN,
E_CLD_BAS_PE_POND,
E_CLD_BAS_PE_RECEPTION_ROOM,
E_CLD_BAS_PE_BREAKFAST_ROOM,
E_CLD_BAS_PE_NOOK,
E_CLD_BAS_PE_GARDEN,
E_CLD_BAS_PE_PANIC_ROOM,
E_CLD_BAS_PE_TERRACE,
E_CLD_BAS_PE_ROOF,
E_CLD_BAS_PE_TOILET,
E_CLD_BAS_PE_TOILET_MAIN,
E_CLD_BAS_PE_OUTSIDE_TOILET,
E_CLD_BAS_PE_SHOWER_ROOM,
E_CLD_BAS_PE_STUDY,
E_CLD_BAS_PE_FRONT_GARDEN,
E_CLD_BAS_PE_BACK_GARDEN,
E_CLD_BAS_PE_KETTLE,
E_CLD_BAS_PE_TELEVISION,
E_CLD_BAS_PE_STOVE,
E_CLD_BAS_PE_MICROWAVE,
```

(continues on next page)

(continued from previous page)

```
    E_CLD_BAS_PE_TOASTER,
    E_CLD_BAS_PE_VACUMM,
    E_CLD_BAS_PE_APPLIANCE,
    E_CLD_BAS_PE_FRONT_DOOR,
    E_CLD_BAS_PE_BACK_DOOR,
    E_CLD_BAS_PE_FRIDGE_DOOR,
    E_CLD_BAS_PE_MEDICATION_CABINET_DOOR,
    E_CLD_BAS_PE_WARDROBE_DOOR,
    E_CLD_BAS_PE_FRONT_CUPBOARD_DOOR,
    E_CLD_BAS_PE_OTHER_DOOR,
    E_CLD_BAS_PE_WAITING_ROOM,
    E_CLD_BAS_PE_TRIAGE_ROOM,
    E_CLD_BAS_PE_DOCTOR_OFFICE,
    E_CLD_BAS_PE_PATIENT_PRIVATE_ROOM,
    E_CLD_BAS_PE_CONSULTATION_ROOM,
    E_CLD_BAS_PE_NURSE_STATION,
    E_CLD_BAS_PE_WARD,
    E_CLD_BAS_PE_COORIDOR,
    E_CLD_BAS_PE_OPERATING_THREATER,
    E_CLD_BAS_PE_DENTAL_SURGERY_ROOM,
    E_CLD_BAS_PE_MEDICAL_IMAGING_ROOM,
    E_CLD_BAS_PE_DECONTAMINATION_ROOM,
    E_CLD_BAS_PE_UNKNOWN_ENVIRONMENT
} teCLD_BAS_PhysicalEnvironment;
```

Parent topic: Enumerations

Parent topic:*Basic Cluster*

Compile-time options To enable the Basic cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define BASIC_CLIENT
#define BASIC_SERVER
```

The Basic cluster contains macros that may be optionally specified at compile-time by adding some or all of the following lines to the **zcl_options.h** file.

Optional Attributes Add this line to enable the optional Application Version attribute:

```
#define   CLD_BAS_ATTR_APPLICATION_VERSION
```

Add this line to enable the optional Stack Version attribute:

```
#define   CLD_BAS_ATTR_STACK_VERSION
```

Add this line to enable the optional Hardware Version attribute:

```
#define CLD_BAS_ATTR_HARDWARE_VERSION
```

Add this line to enable the optional Manufacturer Name attribute:

```
#define CLD_BAS_ATTR_MANUFACTURER_NAME
```

Add this line to enable the optional Model Identifier attribute:

```
#define CLD_BAS_ATTR_MODEL_IDENTIFIER
```

Add this line to enable the optional Date Code attribute:

```
#define CLD_BAS_ATTR_DATE_CODE
```

Add this line to enable the optional Generic Class Device attribute:

```
#define CLD_BAS_ATTR_GENERIC_DEVICE_CLASS
```

Add this line to enable the optional Generic Device Type attribute:

```
#define CLD_BAS_ATTR_GENERIC_DEVICE_TYPE
```

Add this line to enable the optional Product Code attribute:

```
#define CLD_BAS_ATTR_PRODUCT_CODE
```

Add this line to enable the optional Product URL attribute:

```
#define CLD_BAS_ATTR_PRODUCT_URL
```

Add this line to enable the optional Location Description attribute:

```
#define CLD_BAS_ATTR_LOCATION_DESCRIPTION
```

Add this line to enable the optional Physical Environment attribute:

```
#define CLD_BAS_ATTR_PHYSICAL_ENVIRONMENT
```

Add this line to enable the optional Device Enabled attribute:

```
#define CLD_BAS_ATTR_DEVICE_ENABLED
```

Add this line to enable the optional Alarm Mask attribute:

```
#define CLD_BAS_ATTR_ALARM_MASK
```

Add this line to enable the optional Disable Local Config attribute:

```
#define CLD_BAS_ATTR_DISABLE_LOCAL_CONFIG
```

Add this line to enable the optional Software Build ID attribute:

```
#define CLD_BAS_ATTR_SW_BUILD_ID
```

Global Attributes Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_BAS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Optional Commands Add this line to enable the optional Reset To Factory Defaults command on the client and server:

```
#define CLD_BAS_CMD_RESET_TO_FACTORY_DEFAULTS
```

Product Code Length The default length of the product code contained in the attributes sProductCode and au8ProductCode[] can be defined by adding the following line:

```
#define CLD_BAS_PCODE_SIZE <n>
```

where <n> is the default number characters in the product code.

The maximum length of the product code contained in the attributes sProductCode and au8ProductCode[] can be defined by adding the following line:

```
#define CLD_BASIC_MAX_NUMBER_OF_BYTES_PRODUCT_CODE <n>
```

where <n> is the maximum number characters in the product code.

Product URL Length The default length of the product URL contained in the attributes sProductURL and au8ProductURL[] can be defined by adding the following line:

```
#define CLD_BAS_URL_SIZE <n>
```

where <n> is the default number characters in the product URL.

The maximum length of the product URL contained in the attributes sProductURL and au8ProductURL[] can be defined by adding the following line:

```
#define CLD_BASIC_MAX_NUMBER_OF_BYTES_PRODUCT_URL <n>
```

where <n> is the maximum number characters in the product URL.

Parent topic:*Basic Cluster*

Power Configuration Cluster This chapter describes the Power Configuration cluster which is concerned with the power source(s) of a device.

The Power Configuration cluster has a Cluster ID of 0x0001.

Overview The Power Configuration cluster allows:

- information to be obtained about the power source(s) of a device
- voltage alarms to be configured

To use the functionality of this cluster, you must include the file **PowerConfiguration.h** in your application and enable the cluster by defining CLD_POWER_CONFIGURATION in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to change the power configuration on the local device.
- The cluster client is able to send commands to change the power configuration on the remote device.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Power Configuration cluster are fully detailed in Section 9.6.

Parent topic:*Power Configuration Cluster*

Power Configuration Cluster structure and attributes The structure definition for the Power Configuration cluster is:

```
typedef struct
{
#ifdef POWER_CONFIGURATION_SERVER
#ifdef CLD_PWRCFG_ATTR_MAINS_VOLTAGE
    zuint16  u16MainsVoltage;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_FREQUENCY
    zuint8   u8MainsFrequency;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_ALARM_MASK
    zbmap8   u8MainsAlarmMask;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_VOLTAGE_MIN_THRESHOLD
    uint16   u16MainsVoltageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_VOLTAGE_MAX_THRESHOLD
    uint16   u16MainsVoltageMaxThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_VOLTAGE_DWELL_TRIP_POINT
    uint16   u16MainsVoltageDwellTripPoint;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_VOLTAGE
    uint8    u8BatteryVoltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_PERCENTAGE_REMAINING
    uint8    u8BatteryPercentageRemaining;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_MANUFACTURER
    tsZCL_CharacterString   sBatteryManufacturer;
    uint8    au8BatteryManufacturer[16];
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_SIZE
    zenum8   u8BatterySize;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_AHR_RATING
```

(continues on next page)

(continued from previous page)

```
    zuint16       u16BatteryAHRating;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_QUANTITY
    zuint8    u8BatteryQuantity;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_RATED_VOLTAGE
    zuint8    u8BatteryRatedVoltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_ALARM_MASK
    zbmap8    u8BatteryAlarmMask;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_VOLTAGE_MIN_THRESHOLD
    zuint8    u8BatteryVoltageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD1
    zuint8    u8BatteryVoltageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD2
    zuint8    u8BatteryVoltageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD3
    zuint8    u8BatteryVoltageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_MIN_THRESHOLD
    zuint8    u8BatteryPercentageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD1
    zuint8    u8BatteryPercentageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD2
    zuint8    u8BatteryPercentageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD3
    zuint8    u8BatteryPercentageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_ALARM_STATE
    zbmap32       u32BatteryAlarmState;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_VOLTAGE
    uint8     u8Battery2Voltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_PERCENTAGE_REMAINING
    uint8     u8Battery2PercentageRemaining;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_MANUFACTURER
    tsZCL_CharacterString    sBattery2Manufacturer;
    uint8     au8Battery2Manufacturer[16];
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_SIZE
    zenum8    u8Battery2Size;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_AHR_RATING
    zuint16       u16Battery2AHRating;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_QUANTITY
    zuint8    u8Battery2Quantity;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_RATED_VOLTAGE
    zuint8    u8Battery2RatedVoltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_ALARM_MASK
```

(continues on next page)

(continued from previous page)

```
    zbmap8    u8Battery2AlarmMask;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_VOLTAGE_MIN_THRESHOLD
    zuint8    u8Battery2VoltageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD1
    zuint8    u8Battery2VoltageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD2
    zuint8    u8Battery2VoltageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD3
    zuint8    u8Battery2VoltageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_MIN_THRESHOLD
    zuint8    u8Battery2PercentageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD1
    zuint8    u8Battery2PercentageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD2
    zuint8    u8Battery2PercentageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD3
    zuint8    u8Battery2PercentageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_ALARM_STATE
    zbmap32       u32Battery2AlarmState;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_VOLTAGE
    uint8     u8Battery3Voltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_PERCENTAGE_REMAINING
    uint8     u8Battery3PercentageRemaining;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_MANUFACTURER
    tsZCL_CharacterString    sBattery3Manufacturer;
    uint8     au8Battery3Manufacturer[16];
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_SIZE
    zenum8    u8Battery3Size;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_AHR_RATING
    zuint16       u16Battery3AHRating;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_QUANTITY
    zuint8    u8Battery3Quantity;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_RATED_VOLTAGE
    zuint8    u8Battery3RatedVoltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_ALARM_MASK
    zbmap8    u8Battery3AlarmMask;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_VOLTAGE_MIN_THRESHOLD
    zuint8    u8Battery3VoltageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD1
    zuint8    u8Battery3VoltageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD2
```

(continues on next page)

(continued from previous page)

```
    zuint8    u8Battery3VoltageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD3
    zuint8    u8Battery3VoltageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_MIN_THRESHOLD
    zuint8    u8Battery3PercentageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD1
    zuint8    u8Battery3PercentageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD2
    zuint8    u8Battery3PercentageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD3
    zuint8    u8Battery3PercentageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_ALARM_STATE
    zbmap32       u32Battery3AlarmState;
#endif
#ifdef  CLD_PWRCFG_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
    zuint8    u8AttributeReportingStatus;
#endif
#endif
    zuint16       u16ClusterRevision;
} tsCLD_PowerConfiguration;
```

The attributes are classified into four attribute sets: Mains Information, Mains Settings, Battery Information, Battery Settings and Global. The attributes from these sets are described below.

Mains Information Attribute Set

- u16MainsVoltage is the measured AC (RMS) mains voltage or DC voltage currently applied to the device, in units of 100 mV.
- u8MainsFrequency is half of the measured AC mains frequency, in Hertz, currently applied to the device. Actual frequency = 2 x u8MainsFrequency. This allows AC mains frequencies to be stored in the range 2-506 Hz in steps of 2 Hz. In addition:
  - 0x00 indicates a DC supply or that AC frequency is too low to be measured
  - 0xFE indicates that AC frequency is too high to be measured
  - 0xFF indicates that AC frequency could not be measured.

Mains Settings Attribute Set

- u8MainsAlarmMask is a bitmap indicating which mains voltage alarms can be generated (a bit is set to '1' if the alarm is enabled):

Bit	Description
0	Under-voltage alarm (triggered when measured RMS mains voltage falls below a predefined threshold - see below)
1	Over-voltage alarm (triggered when measured RMS mains voltage rises above a predefined threshold - see below)
2	Mains power supply has been lost or is unavailable - that is, the device is now running on battery power.
3-7	Reserved

- u16MainsVoltageMinThreshold is the threshold for the under-voltage alarm, in units of 100 mV. The RMS mains voltage is allowed to fall below this threshold for the duration specified by 16MainsVoltageDwellTripPoint before the alarm is triggered (see below). 0xFFFF indicates that the alarm will not be generated.
- u16MainsVoltageMaxThreshold is the threshold for the over-voltage alarm, in units of 100 mV. The RMS mains voltage is allowed to rise above this threshold for the duration specified by 16MainsVoltageDwellTripPoint before the alarm is triggered (see below). 0xFFFF indicates that the alarm will not be generated.
- u16MainsVoltageDwellTripPoint defines the time-delay, in seconds, before an over-voltage or under-voltage alarm will be triggered when the mains voltage crosses the relevant threshold. If the mains voltage returns within the limits of the thresholds during this time, the alarm will be cancelled. 0xFFFF indicates that the alarms will not be generated.

Battery Information Attribute Set (Battery 1)

- u8BatteryVoltage is the measured battery voltage currently applied to the device, in units of 100 mV. 0xFF indicates that the measured voltage is invalid or unknown.
- u8BatteryPercentageRemaining indicates the remaining battery life as a percentage of the complete battery lifespan, expressed to the nearest half-percent in the range 0 to 100 - for example, 0xAF represents 87.5%. The special value 0xFF indicates an invalid or unknown measurement.

Battery Settings Attribute Set (Battery 1)

- sBatteryManufacturer is a pointer to the array containing the name of the battery manufacturer (see below).
- au8BatteryManufacturer[16] is a 16-element array containing the name of the battery manufacturer (maximum of 16 characters).
- u8BatterySize is an enumeration indicating the type of battery in the device - the enumerations are listed in Section 9.5.2.
- u16BatteryAHRating is the Ampere-hour (Ah) charge rating of the battery, in units of 10 mAh.
- u8BatteryQuantity is the number of batteries used to power the device.
- u8BatteryRatedVoltage is the rated voltage of the battery, in units of

  100 mV.
- u8BatteryAlarmMask is a bitmap indicating whether the battery-low alarm can be generated - if enabled, the alarm is generated when the battery voltage falls below a pre-defined threshold (see below). The alarm-enable bit is bit 0 (which is set to '1' if the alarm is enabled).
- u8BatteryVoltageMinThreshold is the battery voltage threshold, in units of 100 mV, below which the device cannot operate or transmit - a battery-low alarm can be triggered when the battery voltage falls below this threshold:

Value	Description
0x00 - 0x39	Minimum battery voltage threshold, in units of 100 mV
0x3A	Mains power supply has been lost or is unavailable - that is, the device is now running on battery power.
0x3B - 0xFF	Reserved

- u8BatteryVoltageThreshold1is a battery voltage threshold, in units of 100 mV, which can correspond to a battery-low alarm - that is, if the battery voltage falls below this threshold, an alarm can be triggered. It must be greater than the value defined for u8BatteryVoltageMinThreshold. The special value 0xFF indicates that the threshold is not used.u8BatteryVoltageThreshold2 is a battery voltage threshold, in units of 100 mV, which can correspond to a battery-low alarm - that is, if the battery voltage falls below this threshold, an alarm can be triggered. It must be greater than the value defined for u8BatteryVoltageThreshold1. The special value 0xFF indicates that the threshold is not used.
- u8BatteryVoltageThreshold3is a battery voltage threshold, in units of 100 mV, which can correspond to a battery-low alarm - that is, if the battery voltage falls below this threshold, an alarm can be triggered. It must be greater than the value defined for u8BatteryVoltageThreshold2. The special value 0xFF indicates that the threshold is not used.
- u8BatteryPercentageMinThreshold is the minimum alarm threshold for percentage battery-life, expressed in half-percent steps in the range 0 to 100 - if the remaining percentage battery-life (u8BatteryPercentageRemaining) falls below this threshold, an alarm can be triggered.
- u8BatteryPercentageThreshold1 is an alarm threshold for percentage battery-life, expressed in half-percent steps in the range 0 to 100 - if the remaining percentage battery-life (u8BatteryPercentageRemaining) falls below this threshold, an alarm can be triggered. It must be greater than the value defined for u8BatteryPercentageMinThreshold. The special value 0xFF indicates that the threshold is not used.
- u8BatteryPercentageThreshold2is an alarm threshold for percentage battery-life, expressed in half-percent steps in the range 0 to 100 - if the remaining percentage battery-life (u8BatteryPercentageRemaining) falls below this threshold, an alarm can be triggered. It must be greater than the value defined for u8BatteryPercentageThreshold1. The special value 0xFF indicates that the threshold is not used.
- u8BatteryPercentageThreshold3 is an alarm threshold for percentage battery-life, expressed in half-percent steps in the range 0 to 100 - if the remaining percentage battery-life (u8BatteryPercentageRemaining) falls below this threshold, an alarm can be triggered. It must be greater than the value defined for u8BatteryPercentageThreshold2. The special value 0xFF indicates that the threshold is not used.
- u32BatteryAlarmState is a bitmap representing the current state of the alarms for the battery or batteries (the bitmap includes status bits for optional additional batteries 2 and 3). It indicates the state of the battery in relation to the voltage and percentage-life thresholds defined by the attributes above (a bit is set to '1' when the corresponding threshold has been reached).

Bit	Description
Bits for Battery	
0	Bit is set if one of the following thresholds has been reached:

```
u8BatteryVoltageMinThreshold
u8BatteryPercentageMinThreshold
```

| |1|Bit is set if one of the following thresholds has been reached:

```
u8BatteryVoltageThreshold1
u8BatteryPercentageThreshold1
```

| |2|Bit is set if one of the following thresholds has been reached:

```
u8BatteryVoltageThreshold2
u8BatteryPercentageThreshold2
```

| |3|Bit is set if one of the following thresholds has been reached:

```
u8BatteryVoltageThreshold3
u8BatteryPercentageThreshold3
```

| |4 - 9|Reserved| |**Bits for Battery 2 (Optional)**| |10|Bit is set if one of the following thresholds has been reached:

```
u8Battery2VoltageMinThreshold
u8Battery2PercentageMinThreshold
```

| |11|Bit is set if one of the following thresholds has been reached:

```
u8Battery2VoltageThreshold1
u8Battery2PercentageThreshold1
```

| |12|Bit is set if one of the following thresholds has been reached:

```
u8Battery2VoltageThreshold2
u8Battery2PercentageThreshold2
```

| |13|Bit is set if one of the following thresholds has been reached:

```
u8Battery2VoltageThreshold3
u8Battery2PercentageThreshold3
```

| |14 - 19|Reserved| |**Bits for Battery 3 (Optional)**| |20|Bit is set if one of the following thresholds has been reached:

```
u8Battery3VoltageMinThreshold
u8Battery3PercentageMinThreshold
```

| |21|Bit is set if one of the following thresholds has been reached:

```
u8Battery3VoltageThreshold1
u8Battery3PercentageThreshold1
```

| |22|Bit is set if one of the following thresholds has been reached:

```
u8Battery3VoltageThreshold2
u8Battery3PercentageThreshold2
```

| |23|Bit is set if one of the following thresholds has been reached:

```
u8Battery3VoltageThreshold3
u8Battery3PercentageThreshold3
```

| |24 - 29|Reserved| |30|Mains power supply has been lost or is unavailable - that is, the device is now running on battery power| |31|Reserved|

Battery Information and Battery Settings Attribute Sets for Battery <X>

The Battery Information and Battery Settings attribute sets are repeated for up to two further (optional) batteries, denoted 2 and 3. The attributes are as follows, where <X> is 2 or 3, and their definitions are identical to those of the equivalent attributes in the Battery Information and Battery Settings attribute sets described above.

```
u8Battery<X>Voltage
u8Battery<X>PercentageRemaining
```

(continues on next page)

(continued from previous page)

```
au8Battery<X>Manufacturer[16]
u8Battery<X>Size
u16Battery<X>AHRating
u8Battery<X>Quantity
u8Battery<X>RatedVoltage
u8Battery<X>AlarmMask
u8Battery<X>VoltageMinThreshold
u8Battery<X>VoltageThreshold1
u8Battery<X>VoltageThreshold2
u8Battery<X>VoltageThreshold3
u8Battery<X>PercentageMinThreshold
u8Battery<X>PercentageThreshold1
u8Battery<X>PercentageThreshold2
u8Battery<X>PercentageThreshold3
u32Battery<X>AlarmState
```

Global Attribute Set

```
u8AttributeReportingStatus is an optional attribute that should be enabled when attribute report-
ing is used for the cluster (see [Section 2.3.5](../../ZCL_fundamentals/topics/attribute_reporting.md#id_
↪d3e2d682-d5e6-4aa5-870f-4d2e0320637e)). The value of this attribute indicates whether there are at-
tribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved.
↪ This attribute is also described in [Section 2.4](../../ZCL_fundamentals/topics/global_attributes.md#id_
↪8f80e769-8ccd-4b4f-8609-579f4457d527).
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specifica-
tion on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster re-
vision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This at-
tribute is also described in [Section 2.4](../../ZCL_fundamentals/topics/global_attributes.md#id_8f80e769-
↪8ccd-4b4f-8609-579f4457d527).
```

Parent topic:*Power Configuration Cluster*

Attributes for Default Reporting

The following attributes of the Power Configuration cluster can be selected for default reporting:

```
u8BatteryPercentageRemaining
u32BatteryAlarmState
```

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic:*Power Configuration Cluster*

Functions

The below Power Configuration cluster function is provided in the NXP implementation of the ZCL:

- eCLD_PowerConfigurationCreatePowerConfiguration

eCLD_PowerConfigurationCreatePowerConfiguration

```
teZCL_Status eCLD_PowerConfigurationCreatePowerConfiguration(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
```

(continues on next page)

(continued from previous page)

```
void *pvEndPointSharedStructPtr,
uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Power Configuration cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Power Configuration cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Power Configuration cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Power Configuration cluster. This parameter can refer to a pre-filled structure called sCLD_PowerConfiguration which is provided in the **PowerConfiguration.h** file.
- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_PowerConfiguration which defines the attributes of Power Configuration cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Functions

Parent topic:*Power Configuration Cluster*

Enumerations and Defines

teCLD_PWRCFG_AttributeId The following structure contains the enumerations used to identify the attributes of the Power Configuration cluster.

```
typedef enum
{
    /* Mains Information attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_MAINS_VOLTAGE              = 0x0000,
    E_CLD_PWRCFG_ATTR_ID_MAINS_FREQUENCY,
    /* Mains settings attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_MAINS_ALARM_MASK           = 0x0010,
    E_CLD_PWRCFG_ATTR_ID_MAINS_VOLTAGE_MIN_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_MAINS_VOLTAGE_MAX_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_MAINS_VOLTAGE_DWELL_TRIP_POINT,
    /* Battery information attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE            = 0x0020,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_REMAINING,
    /* Battery settings attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_BATTERY_MANUFACTURER       = 0x0030,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_SIZE,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_AHR_RATING,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_QUANTITY,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_RATED_VOLTAGE,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_ALARM_MASK,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_MIN_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD1,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD2,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD3,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_MIN_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD1,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD2,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD3,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_ALARM_STATE,
    /* Battery information 2 attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE          = 0x0040,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_REMAINING,
    /* Battery settings 2 attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_MANUFACTURER     = 0x0050,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_SIZE,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_AHR_RATING,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_QUANTITY,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_RATED_VOLTAGE,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_ALARM_MASK,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_MIN_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD1,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD2,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD3,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_MIN_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD1,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD2,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD3,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_ALARM_STATE,
    /* Battery information 3 attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE          = 0x0060,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_REMAINING,
    /* Battery settings 3 attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_MANUFACTURER     = 0x0070,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_SIZE,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_AHR_RATING,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_QUANTITY,
```

(continues on next page)

(continued from previous page)

```
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_RATED_VOLTAGE,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_ALARM_MASK,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_MIN_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD1,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD2,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD3,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_MIN_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD1,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD2,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD3,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_ALARM_STATE,
    /* Global attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_CLUSTER_REVISION          = 0xFFFC,
    E_CLD_PWRCFG_ATTR_ID_ATTRIBUTE_REPORTING_STATUS = 0xFFFE
} teCLD_PWRCFG_AttributeId;
```

Parent topic: Enumerations and Defines

teCLD_PWRCFG_BatterySize The following structure contains the enumerations used to indicate the type of battery used in the device.

```
typedef enum
{
    E_CLD_PWRCFG_BATTERY_SIZE_NO_BATTERY    = 0x00,
    E_CLD_PWRCFG_BATTERY_SIZE_BUILT_IN,
    E_CLD_PWRCFG_BATTERY_SIZE_OTHER,
    E_CLD_PWRCFG_BATTERY_SIZE_AA,
    E_CLD_PWRCFG_BATTERY_SIZE_AAA,
    E_CLD_PWRCFG_BATTERY_SIZE_C,
    E_CLD_PWRCFG_BATTERY_SIZE_D,
    E_CLD_PWRCFG_BATTERY_SIZE_UNKNOWN       = 0xff,
} teCLD_PWRCFG_BatterySize;
```

Parent topic: Enumerations and Defines

Defines for Voltage Alarms The following #defines are provided for use in the configuration of the mains over-voltage and under-voltage alarms, and the battery-low alarm.

Mains Alarm Mask

```
#define CLD_PWRCFG_MAINS_VOLTAGE_TOO_LOW    (1 << 0)
#define CLD_PWRCFG_MAINS_VOLTAGE_TOO_HIGH   (1 << 1)
```

Battery Alarm Mask

```
#define CLD_PWRCFG_BATTERY_VOLTAGE_TOO_LOW  (1 << 0)
```

Parent topic: Enumerations and Defines

Parent topic:*Power Configuration Cluster*

Compile-time options To enable the Power Configuration cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_POWER_CONFIGURATION
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define POWER_CONFIGURATION_CLIENT
#define POWER_CONFIGURATION_SERVER
```

The Power Configuration cluster contains macros that may be optionally specified at compile-time by adding some or all the following lines to the **zcl_options.h** file.

Optional Attributes Add this line to enable the optional Mains Voltage attribute:

```
#define CLD_PWRCFG_ATTR_MAINS_VOLTAGE
```

Add this line to enable the optional Mains Frequency attribute:

```
#define CLD_PWRCFG_ATTR_MAINS_FREQUENCY
```

Add this line to enable the optional Mains Alarm Mask attribute:

```
#define CLD_PWRCFG_ATTR_MAINS_ALARM_MASK
```

Add this line to enable the optional Mains Voltage Min Threshold attribute:

```
#define CLD_PWRCFG_ATTR_MAINS_VOLTAGE_MIN_THRESHOLD
```

Add this line to enable the optional Mains Voltage Max Threshold attribute:

```
#define CLD_PWRCFG_ATTR_MAINS_VOLTAGE_MAX_THRESHOLD
```

Add this line to enable the optional Mains Voltage Dwell Trip Point attribute:

```
#define CLD_PWRCFG_ATTR_MAINS_VOLTAGE_DWELL_TRIP_POINT
```

Add this line to enable the optional Battery Voltage attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_VOLTAGE
```

Add this line to enable the optional Battery Manufacturer attributes:

```
#define CLD_PWRCFG_ATTR_BATTERY_MANUFACTURER
```

Add this line to enable the optional Battery Size attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_SIZE
```

Add this line to enable the optional Battery Amp Hour attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_AHR_RATING
```

Add this line to enable the optional Battery Quantity attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_QUANTITY
```

Add this line to enable the optional Battery Rated Voltage attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_RATED_VOLTAGE
```

Add this line to enable the optional Battery Alarm Mask attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_ALARM_MASK
```

Add this line to enable the optional Battery Voltage Min Threshold attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_VOLTAGE_MIN_THRESHOLD
```

Add this line to enable the optional Battery Percentage Life Remaining attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_PERCENTAGE_REMAINING
```

Add this line to enable the optional Battery Voltage Threshold 1 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD1
```

Add this line to enable the optional Battery Voltage Threshold 2 attribute:

```
#define LD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD2
```

Add this line to enable the optional Battery Voltage Threshold 3 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD3
```

Add this line to enable the optional Battery Percentage Life Min Threshold attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_MIN_THRESHOLD
```

Add this line to enable the optional Battery Percentage Life Threshold 1 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD1
```

Add this line to enable the optional Battery Percentage Life Threshold 2 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD2
```

Add this line to enable the optional Battery Percentage Life Threshold 3 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD3
```

Add this line to enable the optional Battery Alarm State attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_ALARM_STATE
```

Add this line to enable the optional Battery <X> Voltage attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_VOLTAGE
```

Add this line to enable the optional Battery <X> Percentage Life Remaining attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_PERCENTAGE_REMAINING
```

Add this line to enable the optional Battery <X> Manufacturer attributes:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_MANUFACTURER
```

Add this line to enable the optional Battery <X> Size attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_SIZE
```

Add this line to enable the optional Battery <X> Amp Hour attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_AHR_RATING
```

Add this line to enable the optional Battery <X> Quantity attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_QUANTITY
```

Add this line to enable the optional Battery <X> Rated Voltage attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_RATED_VOLTAGE
```

Add this line to enable the optional Battery <X> Alarm Mask attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_ALARM_MASK
```

Add this line to enable the optional Battery <X> Voltage Min Threshold attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_VOLTAGE_MIN_THRESHOLD
```

Add this line to enable the optional Battery <X> Voltage Threshold 1 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_VOLTAGE_THRESHOLD1
```

Add this line to enable the optional Battery <X> Voltage Threshold 2 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_VOLTAGE_THRESHOLD2
```

Add this line to enable the optional Battery <X> Voltage Threshold 3 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_VOLTAGE_THRESHOLD3
```

Add this line to enable the optional Battery <X> Percentage Life Remaining attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_PERCENTAGE_MIN_THRESHOLD
```

Add this line to enable the optional Battery <X> Percentage Life Threshold 1 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_PERCENTAGE_THRESHOLD1
```

Add this line to enable the optional Battery <X> Percentage Life Threshold 2 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_PERCENTAGE_THRESHOLD2
```

Add this line to enable the optional Battery <X> Percentage Life Threshold 3 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_PERCENTAGE_THRESHOLD3
```

Add this line to enable the optional Battery <X> Alarm State attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_ALARM_STATE
```

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_PWRCFG_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Global Attributes Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_PWRCFG_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Power Configuration Cluster*

Device Temperature Configuration Cluster This chapter describes the Device Temperature Configuration cluster, which is concerned with internal temperature of a device.

The Device Temperature Configuration cluster has a Cluster ID of 0x0002.

- **Overview**
- **Cluster structure and attributes**
- **Functions**
- **Enumerations and Defines**
- **Compile-time options**

Identify Cluster This chapter describes the Identify cluster which allows a device to identify itself (for example, by flashing an LED on the node).

The Identify cluster has a Cluster ID of 0x0003.

Overview The Identify cluster allows the host device to be put into identification mode in which the node highlights itself in some way to an observer (in order to distinguish itself from other nodes in the network). It is recommended that identification mode should involve flashing a light with a period of 0.5 seconds.

To use the functionality of this cluster, you must include the file **Identify.h** in your application and enable the cluster by defining CLD_IDENTIFY in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to start and stop identification mode on the local device.
- The cluster client is able to send the above commands to the server (and therefore control identification mode on the remote device)

The inclusion of the client or server software must be pre-defined in compile-time options of the application. In addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance.

The compile-time options for the Identify cluster are fully detailed in Section 11.9.

Note: The Identify cluster contains optional functionality for the EZ-mode Commissioning module, which is part of the ZigBee Base Device functionality and is described in the *ZigBee Devices User Guide (JNUG3131)*.

Parent topic:*Identify Cluster*

Identify Cluster Structure and Attribute The structure definition for the Identify cluster is:

```
typedef struct
{
#ifdef IDENTIFY_SERVER
    zuint16                u16IdentifyTime;
#ifdef CLD_IDENTIFY_ATTR_COMMISSION_STATE
    zbmap8                  u8CommissionState;
#endif
#endif
    zuint16                u16ClusterRevision;
} tsCLD_Identify;
```

Where:

- u16IdentifyTime is a mandatory attribute specifying the remaining length of time, in seconds, that the device continues in identification mode. Setting the attribute to a non-zero value puts the device into identification mode and the attribute is then decremented every second.
- u8CommissionState is an optional attribute for use with EZ-mode Commissioning (see *Chapter 40*) to indicate the network status and operational status of the node - this information is contained in a bitmap, as follows:

Bits	Description
0	Network State

- 1 if in the correct network (must be 1 if Operational State bit is 1)
- 0 if not in a network, or in a temporary network, or network status is unknown

| |1|Operational State

- 1 if commissioned for operation (Network State bit is set to 1)

- 0 if not commissioned for operation

| |2 - 7|Reserved|

u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in (see Section 2.4).

Parent topic:*Identify Cluster*

Initialization The function **eCLD_IdentifyCreateIdentify()** is used to create an instance of the Identify cluster. This function is called by the initialization function for the host device but can alternatively be used directly by the application in setting up a custom endpoint which supports the Identify cluster (among others).

Parent topic:*Identify Cluster*

Sending Commands The NXP implementation of the ZCL provides functions for sending commands between an Identify cluster client and server.

Starting and Stopping Identification Mode The function **eCLD_IdentifyCommandIdentifyRequestSend()** is used on the cluster client to send a command to the cluster server requesting identification mode to be started or stopped on the server device. The required action is contained in the payload of the command (see Section 11.7.2):

- Setting the payload element *u16IdentifyTime* to a non-zero value has the effect of requesting that the server device enters identification mode for a time (in seconds) corresponding to the specified value.
- Setting the payload element *u16IdentifyTime* to zero has the effect of requesting the immediate termination of any identification mode that is in progress on the server device.

Identification mode can alternatively be started and stopped on a light of a remote node as described in Section 11.4.2.

Parent topic: Sending Commands

Requesting Identification Effects The function **eCLD_IdentifyCommandTriggerEffectSend()** can be used to request a particular identification effect or behavior on a light of a remote node. This function can be used for entering and leaving identification mode instead of **eCLD_IdentifyCommandIdentifyRequestSend()**.

The possible behaviors that can be requested are as follows:

- **Blink:** Light is switched on and then off (once)
- **Breathe**: Light is switched on and off by smoothly increasing and then decreasing its brightness over a 1 second period, and then this process is repeated 15 times.
- **Okay:**
  - Colour light goes green for 1 second.
  - Monochrome light flashes twice in 1 second.
- **Channel change:**
  - Colour light goes orange for 8 seconds.
  - Monochrome light switches to maximum brightness for 0.5 s and then to minimum brightness for 7.5 s.

- **Finish effect:** Current stage of effect is completed and then identification mode is terminated (for example, for the Breathe effect, only the current 1 second cycle is completed).
- **Stop effect**: Current effect and identification mode are terminated as soon as possible.

Parent topic: Sending Commands

Inquiring about Identification Mode The function **eCLD_IdentifyCommandIdentifyQueryRequestSend()** is called on an Identify cluster client in order to request a response from a server cluster when it is in identification mode. This request should only be unicast.

Parent topic: Sending Commands

Using EZ-mode Commissioning Features The Identify cluster also contains the following optional features that can be used with EZ-mode commissioning, which is a part of the ZigBee Base Device functionality and is described in the *ZigBee Devices User Guide (JNUG3131)*.

'EZ-mode Invoke' Command The 'EZ-mode Invoke' command is supported which allows a device to schedule and start one or more stages of EZ-mode commissioning on a remote device. The command is issued by calling the **eCLD_IdentifyEZModeInvokeCommandSend()** function and allows the following stages to be specified:

1. **Factory Reset:** EZ-mode commissioning configuration of the destination device to be reset to 'Factory Fresh' settings.
2. **Network Steering:** Destination device to be put into the 'Network Steering' phase.
3. **Find and Bind:** Destination device to be put into the 'Find and Bind' phase.

On receiving the command, the event E_CLD_IDENTIFY_CMD_EZ_MODE_INVOKE is generated on the remote device, indicating one or more requested commissioning actions. The local application must perform these actions using the functions of the EZ-mode Commissioning module. If more than one stage is specified, they must be performed sequentially in the above order and must be contiguous.

If the 'EZ-mode Invoke' command is to be used by an application, its use must be enabled at compile time (see Section 11.9).

'Commissioning State' Attribute The Identify cluster server contains an optional 'Commissioning State' attribute, u8CommissionState (see Section 11.2), which indicates whether the local device is:

- a member of the (correct) network
- in a commissioned state and ready for operation

If the 'Commissioning State' attribute is to be used by an application, its use must be enabled at compile time (see Section 11.9).

The EZ-mode initiator can send an 'Update Commission State' command to the target device in order to update the commissioning state of the target. The command is issued by calling the **eCLD_IdentifyUpdateCommissionStateCommandSend()** function. On receiving this command on the target, the 'Commissioning State' attribute is automatically updated. It is good practice for the EZ-mode initiator to send this command to notify the target device when commissioning is complete.

Parent topic: Sending Commands

Parent topic:*Identify Cluster*

Sleeping Devices in Identification Mode In some cases, a device might sleep between activities (for example, a switch that is configured as a sleeping End Device) and is also operating in identification mode. In such a case, the device must wake once per second for the ZCL to decrement the *u16IdentifyTime* attribute (see Section 11.2), which represents the time remaining in identification mode. The device may also use this wake time to highlight itself, for example, flash an LED. The attribute is automatically updated by the ZCL when the application passes an E_ZCL_CBET_TIMER event to the ZCL via the **vZCL_EventHandler()** function. The ZCL also automatically increments ZCL time as a result of this event.

When in identification mode, it is not permissible for a device to sleep for longer than 1 second, and to generate one timer event on waking. Before entering sleep, the value of the *u16IdentifyTime* attribute can be checked. If this value is zero, the device is not in identification mode and is therefore allowed to sleep for longer than 1 second.

For details of updating ZCL time following a prolonged sleep, refer to Section 18.4.1.

Parent topic:*Identify Cluster*

Functions The following Identify cluster functions are provided in the NXP implementation of the ZCL:

- eCLD_IdentifyCreateIdentify
- eCLD_IdentifyCommandIdentifyRequestSend
- eCLD_IdentifyCommandTriggerEffectSend
- eCLD_IdentifyCommandIdentifyQueryRequestSend
- eCLD_IdentifyEZModeInvokeCommandSend
- eCLD_IdentifyUpdateCommissionStateCommandSend

eCLD_IdentifyCreateIdentify

```
teZCL_Status eCLD_IdentifyCreateIdentify(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits,
        tsCLD_IdentifyCustomDataStructure
    *psCustomDataStructure);
```

Description This function creates an instance of the Identify cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function must be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates an Identify cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function should not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be the first Identify cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the

total number of attributes supported by the Identify cluster. The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Identify cluster. This parameter can refer to a pre-filled structure called sCLD_Identify which is provided in the **Identify.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_Identify which defines the attributes of Identify cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
- *psCustomDataStructure*: Pointer to the structure that contains custom data for the Identify cluster (see Section 11.7.1). This structure is used for internal data storage. No knowledge of the fields of this structure is required.

Returns

- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_SUCCESS

Parent topic: Functions

eCLD_IdentifyCommandIdentifyRequestSend

```
teZCL_Status eCLD_IdentifyCommandIdentifyRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_Identify_IdentifyRequestPayload *psPayload);
```

Description This function can be called on a client device to send a custom command requesting that the recipient server device either enters or exits identification mode. The required action (start or stop identification mode) must be specified in the payload of the custom command (see Section 11.7.2). The required duration of the identification mode is specified in the payload and this value replaces the value in the Identify cluster structure on the target device.

A device which receives this command generates a callback event on the endpoint on which the Identify cluster was registered.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 11.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IdentifyCommandTriggerEffectSend

```
teZCL_Status eCLD_IdentifyCommandTriggerEffectSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    teCLD_Identify_EffectId eEffectId,
    uint8 u8EffectVariant);
```

Description This function can be called on a client device to send a custom command to a server device, in order to control the identification effect on a light of the target node. Therefore, this function can be used to start and stop identification mode instead of **eCLD_IdentifyCommandIdentifyRequestSend()**. Use of the 'Trigger Effect' function must be enabled via a compile-time option, as described in Section 11.9.

The following effect commands can be sent using this function:

Effect command	Description
Blink	Light is switched on and then off (once)
Breathe	Light is switched on and off by smoothly increasing and then decreasing its brightness over a 1-second period, and then this is repeated 15 times
Okay	- Color light goes green for 1 second

- Monochrome light flashes twice in 1 second

| |Channel change|- Color light goes orange for 8 seconds

- Monochrome light switches to

maximum brightness for 0.5 s and then to

minimum brightness for 7.5 s

| |Finish effect|Current stage of effect is completed and then identification mode is terminated (for example, for the Breathe effect, only the current 1-second cycle is completed)| |Stop effect|Current effect and identification mode are terminated as soon as possible|

A variant of the selected effect can also be specified, but currently only the default (as described above) is available.

A device which receives this command generates a callback event on the endpoint on which the Identify cluster was registered.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *eEffectId*: Effect command to send (see above), one of:
  - E_CLD_IDENTIFY_EFFECT_BLINK
  - E_CLD_IDENTIFY_EFFECT_BREATHE
  - E_CLD_IDENTIFY_EFFECT_OKAY
  - E_CLD_IDENTIFY_EFFECT_CHANNEL_CHANGE
  - E_CLD_IDENTIFY_EFFECT_FINISH_EFFECT
  - E_CLD_IDENTIFY_EFFECT_STOP_EFFECT
- *u8EffectVariant*: Required variant of specified effect - set to zero for default (as no variants are currently available).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IdentifyCommandIdentifyQueryRequestSend

```
tsZCL_Status eCLD_IdentifyCommandIdentifyQueryRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be called on a client device to send a custom command requesting a response from any server devices that are currently in identification mode.

A device which receives this command generates a callback event on the endpoint on which the Identify cluster is registered. If the receiving device is in identification mode, it returns a response containing the amount of time for which it continues in this mode (see Section 11.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IdentifyEZModeInvokeCommandSend

```
teZCL_Status eCLD_IdentifyEZModeInvokeCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    bool bDirection,
    tsCLD_Identify_EZModeInvokePayload
    *psPayload);
```

Description This function can be used to send an 'EZ-mode Invoke' to a remote device. The sent command requests one or more of the following stages of the EZ-mode commissioning process to be performed on the destination device. EZ-mode commissioning is a part of the ZigBee Base Device functionality and is described in the *ZigBee Devices User Guide (JNUG3131)*.

1. Factory Reset - clears all bindings, group table entries, and the u8CommissionState attribute, and reverts to the 'Factory Fresh' settings.
2. Network Steering - puts the destination device into the 'Network Steering' phase.
3. Find and Bind - puts the destination device into the 'Find and Bind' phase.

The required stages are specified in a bitmap in the command payload structure tsCLD_Identify_EZModeInvokePayload (see Section 11.7.4). If more than one stage is specified, they must be performed in the above order and be contiguous.

On receiving the 'EZ-mode Invoke' command on the destination device, an E_CLD_IDENTIFY_CMD_EZ_MODE_INVOKE event is generated with the required commissioning actions specified in the u8Action field of the tsCLD_Identify_EZModeInvokePayload structure. It is the responsibility of the local application to perform the requested actions using the functions of the EZ-mode Commissioning module (see Section 40.6).

Note that the 'EZ-mode Invoke' command is optional and, if necessary, must be enabled in the compile-time options (see Section 11.9).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.

- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *bDirection*: Boolean indicating the direction of the command, as follows (this should always be set to TRUE):
  - TRUE - Identify cluster client to server
  - FALSE - Identify cluster server to client
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 11.7.4).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IdentifyUpdateCommissionStateCommandSend

```
teZCL_Status eCLD_IdentifyUpdateCommissionStateCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_Identify_UpdateCommissionStatePayload
*psPayload);
```

Description This function can be used to send an 'Update Commission State' command from an EZ-mode initiator device (cluster client) to a target device (cluster server) in order to update the (optional) u8CommissionState attribute (see Section 11.2) which is used for EZ-mode commissioning (which is part of the ZigBee Base Device functionality and is described in the *ZigBee Devices User Guide (JNUG3131)*). The command allows individual bits of u8CommissionState to be set or cleared (see Section 11.7.4).

On receiving the 'Update Commission State' command on the target device, an event is generated and the requested update is automatically performed.

Note that the u8CommissionState attribute is optional and, if necessary, must be enabled in the compile-time options (see Section 11.9).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 11.7.4).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

Parent topic:*Identify Cluster*

Structures

Custom Data Structure The Identity cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress                    sReceiveEventAddress;
    tsZCL_CallBackEvent                              sCustomCallBackEvent;
    tsCLD_IdentifyCallBackMessage          sCallBackMessage;
        } tsCLD_IdentifyCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Custom Command Payloads The following structure contains the payload for an Identify cluster custom command (sent using the function **eCLD_IdentifyCommandIdentifyRequestSend()**):

```
/* Identify request command payload */
typedef struct
{
    zuint16                u16IdentifyTime;
} tsCLD_Identify_IdentifyRequestPayload;
```

where u16IdentifyTime is the amount of time, in seconds, for which the target device is to remain in identification mode. If this element is set to 0x0000 and the target device is in identification mode, the mode is terminated immediately.

Parent topic: Structures

Custom Command Responses The following structure contains the response to a query whether a device is in identification mode (the original query is sent using the function **eCLD_IdentifyCommandIdentifyQueryRequestSend()**):

```
/* Identify query response command payload */
typedef struct
{
    zuint16                u16Timeout;
} tsCLD_Identify_IdentifyQueryResponsePayload;
```

where u16Timeout is the amount of time, in seconds, that the responding device remains in identification mode.

Parent topic: Structures

EZ-mode Commissioning Command Payloads The structures described below may be used when the Identify cluster is used with EZ-mode commissioning (which is part of the ZigBee Base Device functionality and is described in the *ZigBee Devices User Guide (JNUG3131)*).

'EZ-Mode Invoke' Command Payload The following structure is used when sending an 'EZ-mode Invoke' command (using the **eCLD_IdentifyEZModeInvokeCommandSend()** function).

```
typedef struct
{
    zbmap8     u8Action;
} tsCLD_Identify_EZModeInvokePayload;
```

where u8Action is a bitmap specifying the EZ-mode commissioning actions to be performed on the destination device - a bit is set to '1' if the corresponding action is required, or to '0' if it is not required:

Bits	Action
0	Factory Reset - clears all bindings, group table entries, and the u8CommissionState attribute, and reverts to the 'Factory Fresh' settings
1	Network Steering - puts the device into the 'Network Steering' phase
2	Find and Bind - puts the device into the 'Find and Bind' phase
3 - 7	Reserved

'Update Commission State' Command Payload The following structure is used when sending an 'Update Commission State' command (using the **eCLD_IdentifyUpdateCommissionStateCommandSend()** function), which requests an update to the value of the u8CommissionState attribute (for the definition of the attribute, refer to Section 11.2).

```
typedef struct
{
    zenum8    u8Action;
    zbmap8    u8CommissionStateMask;
} tsCLD_Identify_UpdateCommissionStatePayload;
```

where:

- u8Action is a value specifying the action to perform (set or clear) on the u8CommissionState bits specified through u8CommissionStateMask:
  - 1: Set the specified bits to '1'.
  - 2: Clear the specified bits to '0'.
    - All other values are reserved.
- u8CommissionStateMask is a bitmap in which the bits correspond to the bits of the u8CommissionState attribute. A bit of this field indicates whether the corresponding attribute bit is to be updated (according to the action specified in u8Action):
  - If a bit is set to '1', the corresponding u8CommissionState bit should be updated.
  - If a bit is set to '0', the corresponding u8CommissionState bit should not be updated.

Parent topic: Structures

Parent topic:*Identify Cluster*

Enumerations

teCLD_Identify_ClusterID The following structure contains the enumerations used to identify the attributes of the Identify cluster.

```
typedef enum
{
    E_CLD_IDENTIFY_ATTR_ID_IDENTIFY_TIME = 0x0000,    /* Mandatory */
    E_CLD_IDENTIFY_ATTR_ID_COMMISSION_STATE           /* Optional */
 } teCLD_Identify_ClusterID;
```

Parent topic: Enumerations

Parent topic:*Identify Cluster*

Compile-time options To enable the Identify cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_IDENTIFY
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define IDENTIFY_CLIENT
#define IDENTIFY_SERVER
```

The following cluster functionality can be enabled or configured in **zcl_options.h**.

Cluster Revision Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_IDENTIFY_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Trigger Effect Add this line to enable use of the **eCLD_IdentifyCommandTriggerEffectSend()** function to remotely start/stop identification mode:

```
#define CLD_IDENTIFY_CMD_TRIGGER_EFFECT
```

Enhanced Functionality for EZ-mode Commissioning To enable the optional 'Commission State' attribute, you must include:

```
#define CLD_IDENTIFY_ATTR_COMMISSION_STATE
```

To enable the optional 'EZ-mode Invoke' command, you must include:

```
#define CLD_IDENTIFY_CMD_EZ_MODE_INVOKE
```

EZ-mode commissioning is part of the ZigBee Base Device functionality and is described in the *ZigBee Devices User Guide (JNUG3131)*.

Parent topic:*Identify Cluster*

Groups Cluster This chapter describes the Groups cluster which allows the management of the Group table concerned with group addressing.

The Groups cluster has a Cluster ID of 0x0004.

Overview The Groups cluster allows the management of group addressing that is available in ZigBee PRO. In this addressing scheme, an endpoint on a device can be a member of a group comprising endpoints from one or more devices. The group is assigned a 16-bit group ID or address. The group ID and the local member endpoint numbers are held in an entry of the Group table on a device. If a message is sent to a group address, the Group table is used to determine to which endpoints (if any) the message should deliver on the device. A group can be assigned a name of up to 16 characters and the cluster allows the support of group names to be enabled/disabled.

To use the functionality of this cluster, you must include the file **Groups.h** in your application and enable the cluster by defining CLD_GROUPS in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to modify the local group table.

- The cluster client is able to send commands to the server to request changes to the group table on the server.

The inclusion of the client or server software must be pre-defined in the compile-time options of the application. In addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance.

The compile-time options for the Groups cluster are fully detailed in Section 12.8.

Parent topic:*Groups Cluster*

Groups Cluster structure and attributes The structure definition for the Groups cluster is:

```
typedef struct
{
    zbmap8              u8NameSupport;
    zuint16             u16ClusterRevision;
} tsCLD_Groups;
```

where:

- u8NameSupport indicates whether group names are supported by the cluster:
  - A most significant bit of 1 indicates that group names are supported.
  - A most significant bit of 0 indicates that group names are not supported.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Groups Cluster*

Initialization The function **eCLD_GroupsCreateGroups()** is used to create an instance of the Groups cluster. The function is called by the initialization function for the host device.

A local endpoint can be added to a group on the local node using the function **eCLD_GroupsAdd()**. If the group does not exist, the function creates it. Therefore, this is a way of creating a local group.

Parent topic:*Groups Cluster*

Sending Commands The NXP implementation of the ZCL provides functions for sending commands between a Groups cluster client and server. A command is sent from the client to one or more endpoints on the server. Multiple endpoints can be targeted using binding or group addressing.

Adding Endpoints to Groups Two functions are provided for adding one or more endpoints to a group on a remote device. Each function sends a command to the endpoints to be added to the group, where the required group is specified in the payload of the command. If the group does not exist in the Group table of the target device, it is added to the table.

- **eCLD_GroupsCommandAddGroupRequestSend()** can be used to request the addition of the target endpoints to the specified group.
- **eCLD_GroupsCommandAddGroupIfIdentifyingRequestSend()** can be used to request the addition of the target endpoints to the specified group if the target device is in identification mode of the Identity cluster (see *Chapter 11*).

An endpoint can also be added to a local group, as described in Section 12.3.

Parent topic: Sending Commands

Removing Endpoints from Groups Two functions are provided for removing one or more endpoints from groups on a remote device. Each function sends a command to the endpoints to be removed from the groups. If a group is empty following the removal of the endpoint, it is deleted in the Group table.

- **eCLD_GroupsCommandRemoveGroupRequestSend()** can be used to request the removal of the target endpoint from the group which is specified in the payload of the command.
- **eCLD_GroupsCommandRemoveAllGroupsRequestSend()** can be used to request the removal of the target endpoint from all groups on the remote device.

If an endpoint is a member of a scene associated with a group to be removed, the above function calls also results in the removal of the endpoint from the scene.

Parent topic: Sending Commands

Obtaining Information about Groups Two functions are provided for obtaining information about groups. Each function sends a command to the endpoints to which the inquiry relates.

- **eCLD_GroupsCommandViewGroupRequestSend()** can be used to request the name of a group with the ID/address specified in the command payload.
- **eCLD_GroupsCommandGetGroupMembershipRequestSend()** can be used to determine whether the target endpoint is a member of any of the groups specified in the command payload.

Parent topic: Sending Commands

Parent topic:*Groups Cluster*

Functions The following Groups cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_GroupsCreateGroups
2. eCLD_GroupsAdd
3. eCLD_GroupsCommandAddGroupRequestSend
4. eCLD_GroupsCommandViewGroupRequestSend
5. eCLD_GroupsCommandGetGroupMembershipRequestSend
6. eCLD_GroupsCommandRemoveGroupRequestSend
7. eCLD_GroupsCommandRemoveAllGroupsRequestSend
8. eCLD_GroupsCommandAddGroupIfIdentifyingRequestSend

eCLD_GroupsCreateGroups

```
teZCL_Status eCLD_GroupsCreateGroups(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    tsCLD_GroupsCustomDataStructure
    *psCustomDataStructure,
```

(continues on next page)

(continued from previous page)

```
tsZCL_EndPointDefinition *psEndPointDefinition);
```

Description This function creates an instance of the Groups cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function must be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates a Groups cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: Do not call this function for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be the first Groups cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function retrieves any group IDs already stored in the Application Information Base (AIB) of the ZigBee PRO stack. However, the AIB does not store group names. If name support is required, the application should store the group names using the NVM module, so that they can be retrieved following a power outage.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Groups cluster. This parameter can refer to a pre-filled structure called sCLD_Groups which is provided in the **Groups.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_Groups which defines the attributes of Groups cluster. The function initializes the attributes with default values.
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see [Section 12.6.1](custom_data_structure.md #id_1d50fd17-874c-48c4-b748-e26925fd08d3)).
- *psEndPointDefinition*: Pointer to the ZCL endpoint definition structure for the application (see Section 6.1.1).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Functions

eCLD_GroupsAdd

```
teZCL_Status eCLD_GroupsAdd(uint8 u8SourceEndPointId,
    uint16 u16GroupId,
    uint8 *pu8GroupName);
```

Description This function adds the specified endpoint on the local node to the group with the specified group ID/address and specified group name. The relevant entry is modified in the Group table on the local endpoint (of the calling application). If the group does not currently exist, it is created by adding a new entry for the group to the Group table.

Note that the number of entries in the Group table must not exceed the value of CLD_GROUPS_MAX_NUMBER_OF_GROUPS defined at compile time (see Section 12.8).

Parameters

- *u8SourceEndPointId*: Number of local endpoint to be added to group.
- *u16GroupId*: 16-bit group ID/address of group.
- *pu8GroupName*: Pointer to character string representing name of group.

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_GroupsCommandAddGroupRequestSend

```
teZCL_Status eCLD_GroupsCommandAddGroupRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_Groups_AddGroupRequestPayload
    *psPayload);
```

Description This function sends an Add Group command to a remote device, requesting that the specified endpoints on the target device be added to a group. The group ID/address and name (if supported) are specified in the payload of the message, and must be added to the Group table on the target node along with the associated endpoint numbers.

The device receiving this message generates a callback event on the endpoint on which the Groups cluster is registered. Also, add the group to its Group table before sending a response indicating success or failure (see Section 12.6.4).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 12.6.3).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_GroupsCommandViewGroupRequestSend

```
teZCL_Status eCLD_GroupsCommandViewGroupRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_Groups_ViewGroupRequestPayload
    *psPayload);
```

Description This function sends a View Group command to a remote device, requesting the name of the group with the specified group ID (address) on the destination endpoint.

The device receiving this message generates a callback event on the endpoint on which the Groups cluster was registered and generates a View Group response containing the group name (see Section 12.6.4).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 12.6.3).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_GroupsCommandGetGroupMembershipRequestSend

```
teZCL_Status eCLD_GroupsCommandGetGroupMembershipRequestSend
    (uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_Groups_GetGroupMembershipRequestPayload
    *psPayload);
```

Description This function sends a Get Group Membership command to inquire whether the target endpoint is a member of any of the groups specified in a list contained in the command payload.

The device receiving this message generates a callback event on the endpoint on which the Groups cluster is registered and generates a Get Group Membership response containing the required information (see Section 12.6.4).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 12.6.3).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_GroupsCommandRemoveGroupRequestSend

```
teZCL_Status eCLD_GroupsCommandRemoveGroupRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_Groups_RemoveGroupRequestPayload
    *psPayload);
```

Description This function sends a Remove Group command to request that the target device deletes membership of the destination endpoints from a particular group - that is, remove the endpoints from the entry of the group in the Group table on the device and, if no other endpoints remain in the group, remove the group from the table.

The device receiving this message generates a callback event on the endpoint on which the Groups cluster is registered. If the group becomes empty following the deletion, the device removes the group ID and group name from its Group table. It then generates an appropriate Remove Group response indicating success or failure (see Section 12.6.4).

If the target endpoint belongs to a scene associated with the group to be removed (requiring the Scenes cluster - see *Chapter 13*), the endpoint is also removed from this scene as a result of this function call - that is, the relevant scene entry is deleted from the Scene table on the target device.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: The number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 12.6.3).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_GroupsCommandRemoveAllGroupsRequestSend

```
teZCL_Status eCLD_GroupsCommandRemoveAllGroupsRequestSend
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function sends a **Remove All Groups** command to request that the target device removes all group memberships of the destination endpoints. Issuing this command removes the endpoints from all group entries in the Group table on the device. If no other endpoints remain in a group, the function removes the group from the table.

The device receiving this message generates a callback event on the endpoint on which the Groups cluster is registered. If a group becomes empty following the deletion, the device removes the group ID and group name from its Group table.

If the target endpoint belongs to scenes associated with the groups to be removed, calling this function also removes the endpoint from the scenes. The relevant scene entries are deleted from the Scene table on the target device. (For details about the Scenes cluster - refer to *Chapter 13*).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: The number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

The eCLD_GroupsCommandRemoveAllGroupsRequestSend function invokes the ZigBee PRO stack function to transmit the data. If the ZigBee PRO stack function returns an error, the same can be obtained by calling the **eZCL_GetLastZpsError()** function.

Parent topic: Functions

eCLD_GroupsCommandAddGroupIfIdentifyingRequestSend

```
teZCL_Status eCLD_GroupsCommandAddGroupIfIdentifyingRequestSend
    (uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_Groups_AddGroupRequestPayload
```

(continues on next page)

(continued from previous page)

```
*psPayload);
```

Description This function sends an Add Group If Identifying command to a remote device, requesting that the specified endpoints on the target device be added to a particular group on the condition that the remote device is identifying itself. The group ID/address and name (if supported) are specified in the payload of the message, and must be added to the Group table on the target node along with the associated endpoint numbers. The identifying functionality is controlled using the Identify cluster (see *Chapter 11*).

The device receiving this message generates a callback event on the endpoint on which the Groups cluster is registered and then checks whether the device is identifying itself. If so, the device (if possible) adds the group ID and group name to its Group table. If the device is not identifying itself, then no action is taken.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 12.6.3).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function, which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

Parent topic:*Groups Cluster*

Structures

Custom Data Structure The Groups cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    DLIST                    lGroupsAllocList;
    DLIST                    lGroupsDeAllocList;
    bool                     bIdentifying;
    tsZCL_ReceiveEventAddress    sReceiveEventAddress;
    tsZCL_CallBackEvent          sCustomCallBackEvent;
    tsCLD_GroupsCallBackMessage  sCallBackMessage;
#if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
    tsCLD_GroupTableEntry
        asGroupTableEntry[CLD_GROUPS_MAX_NUMBER_OF_GROUPS];
#endif
} tsCLD_GroupsCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

However, the structure tsCLD_GroupTableEntry used for the Group table entries is shown in Section 12.6.2.

Parent topic: Structures

Group Table Entry The following structure contains a Group table entry.

```
typedef struct
{
    DNODE    dllGroupNode;
    uint16   u16GroupId;
    uint8    au8GroupName[CLD_GROUPS_MAX_GROUP_NAME_LENGTH + 1];
} tsCLD_GroupTableEntry;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Custom Command Payloads The following structures contain the payloads for the Groups cluster custom commands.

Add Group Request Payload

```
typedef struct
{
    zuint16                  u16GroupId;
    tsZCL_CharacterString    sGroupName;
} tsCLD_Groups_AddGroupRequestPayload;
```

where:

- u16GroupId is the ID/address of the group to which the endpoints must be added.
- sGroupName is the name of the group to which the endpoints must be added.

View Group Request Payload

```
typedef struct
{
    zuint16             u16GroupId;
} tsCLD_Groups_ViewGroupRequestPayload;
```

where u16GroupId is the ID/address of the group whose name is required

Get Group Membership Request Payload

```
typedef struct
{
    zuint8              u8GroupCount;
    zint16              *pi16GroupList;
} tsCLD_Groups_GetGroupMembershipRequestPayload;
```

where:

- u8GroupCount is the number of groups in the list of the next field.
- pi16GroupList is a pointer to a list of groups whose memberships are being queried, where each group is represented by its group ID/address.

Remove Group Request Payload

```
typedef struct
{
    zuint16             u16GroupId;
} tsCLD_Groups_RemoveGroupRequestPayload;
```

where u16GroupId is the ID/address of the group from which the endpoints must be removed.

Parent topic: Structures

Custom Command Responses The Groups cluster generates responses to certain custom commands. The responses which contain payloads are detailed below:

Add Group Response Payload

```
typedef struct
{
    zenum8              eStatus;
    zuint16             u16GroupId;
} tsCLD_Groups_AddGroupResponsePayload;
```

where:

- eStatus is the status (success or failure) of the requested group addition.
- u16GroupId is the ID/address of the group to which endpoints were added.

View Group Response Payload

```
typedef struct
{
    zenum8                  eStatus;
    zuint16                 u16GroupId;
    tsZCL_CharacterString   sGroupName;
} tsCLD_Groups_ViewGroupResponsePayload;
```

where:

- eStatus is the status (success or failure) of the requested operation.
- u16GroupId is the ID/address of the group whose name was requested.
- sGroupName is the returned name of the specified group.

Get Group Membership Response Payload

```
typedef struct
{
    zuint8                  u8Capacity;
    zuint8                  u8GroupCount;
    zint16                  *pi16GroupList;
} tsCLD_Groups_GetGroupMembershipResponsePayload;
```

where:

- u8Capacity is the capacity of the Group table of the device to receive more groups - that is, the number of groups that may be added (special values: 0xFE means that at least one additional group may be added, a higher value means that the remaining capacity of the table is unknown).
- u8GroupCount is the number of groups in the list of the next field.
- pi16GroupList is a pointer to the returned list of groups from those queried that exist on the device, where each group is represented by its group ID/address.

Remove Group Response Payload

```
typedef struct
{
    zenum8                  eStatus;
    zuint16                 u16GroupId;
} tsCLD_Groups_RemoveGroupResponsePayload;
```

where:

- eStatus is the status (success or failure) of the requested group modification.
- u16GroupId is the ID/address of the group from which endpoints were removed.

Parent topic: Structures

Parent topic:*Groups Cluster*

Enumerations

teCLD_Groups_ClusterID The following structure contains the enumeration used to identify the attribute of the Groups cluster.

```
typedef enum
{
   E_CLD_GROUPS_ATTR_ID_NAME_SUPPORT        = 0x0000   /* Mandatory */
} teCLD_Groups_ClusterID;
```

Parent topic: Enumerations

Parent topic:*Groups Cluster*

Compile-time Options To enable the Groups cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_GROUPS
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define GROUPS_CLIENT
#define GROUPS_SERVER
```

The Groups cluster contains macros that may be optionally specified at compile time by adding one or both of the following lines to the **zcl_options.h** file.

To set the size used for the group addressing table in the **.zpscfg** file,

Add this line:

```
#define CLD_GROUPS_MAX_NUMBER_OF_GROUPS          (8)
```

To configure the maximum length of the group name, add the following line:

```
#define CLD_GROUPS_MAX_GROUP_NAME_LENGTH          (16)
```

To define the value (n) of the Cluster Revision attribute, add the following line:

```
#define CLD_GROUPS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Groups Cluster*

Scenes Cluster This chapter describes the Scenes cluster that allows scenes to be managed.

The Scenes cluster has a Cluster ID of 0x0005.

Overview A scene is a set of stored attribute values for one or more cluster instances, where these cluster instances may exist on endpoints on one or more devices.

The Scenes cluster allows standard values for these attributes to be set and retrieved. Thus, the cluster can be used to put the network or part of the network into a pre-defined mode (for example, Night or Day mode for a lighting network). These pre-defined scenes can be used as a basis for 'mood lighting'. A Scenes cluster instance must be created on each endpoint which contains a cluster that is part of a scene.

A scene is often associated with a group (which collects together a set of endpoints over one or more devices) - groups are described in *Chapter 12*. A scene may, however, be used without a group.

Note: When the Scenes cluster is used on an endpoint, a Groups cluster instance must always be created on the same endpoint, even if a group is not used for the scene.

If a cluster on a device is used in a scene, an entry for the scene must be contained in the Scene table on the device. A Scene table entry includes the scene ID, the group ID associated with the scene (0x0000 if there is no associated group), the scene transition time (amount of time to switch to the scene), and the attribute settings for the clusters on the device. The scene ID must be unique within the group with which the scene is associated.

To use the functionality of this cluster, you must include the file **Scenes.h** in your application and enable the cluster by defining CLD_SCENES in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to access scenes.
- The cluster client is able to send commands to the server to request read or write access to scenes.

The inclusion of the client or server software must be pre-defined in the compile-time options of the application (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Scenes cluster are fully detailed in Section 13.9.

Parent topic:*Scenes Cluster*

Scenes Cluster structure and attributes

The structure definition for the Scenes cluster is:

```
typedef struct
{
#ifdef SCENES_SERVER
   zuint8  u8SceneCount;
   zuint8  u8CurrentScene;
   zuint16 u16CurrentGroup;
   zbool   bSceneValid;
   zuint8  u8NameSupport;
#ifdef CLD_SCENES_ATTR_LAST_CONFIGURED_BY
   zieeeaddress  u64LastConfiguredBy;
#endif
#endif
    zuint16 u16ClusterRevision;
} tsCLD_Scenes;
```

where:

- u8SceneCount is the number of scenes currently in the Scene table.
- u8CurrentScene is the scene ID of the last scene invoked on the device.
- u16CurrentGroup is the group ID of the group associated with the last scene invoked (or 0x0000 if this scene is not associated with a group).
- bSceneValid indicates whether the current state of the device corresponds to the values of the CurrentScene and CurrentGroup attributes (TRUE if they do, FALSE if they do not).
- u8NameSupport indicates whether scene names are supported - if the most significant bit is 1 then they are supported, otherwise they are not supported.

- u64LastConfiguredBy is the 64-bit IEEE address of the device that last configured the Scene table (0xFFFFFFFFFFFFFFFF indicates that the address is unknown or the table has not been configured).
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Scenes Cluster*

Initialization The function **eCLD_ScenesCreateScenes()** is used to create an instance of the Scenes cluster. The function is generally called by the initialization function for the host device.

Parent topic:*Scenes Cluster*

Sending Remote Commands The NXP implementation of the ZCL provides functions for sending commands between a Scenes cluster client and server. A command is sent from the client to one or more endpoints on the server. Multiple endpoints can usually be targeted using binding or group addressing.

Note: Commands can also be issued for operations on the local node, as described in Section 13.5.

Creating a Scene In order to create a scene, add an entry for the scene to the Scene table on every device that contains a cluster, which is associated with the scene.

Use the function **eCLD_ScenesCommandAddSceneRequestSend()** to request a scene to be added to a Scene table on a remote device. Invoking this function sends a request to a single device or to multiple devices (using binding or group addressing). The fields of the Scene table entry are specified in the payload of the request.

Alternatively:

- The function **eCLD_ScenesCommandEnhancedAddSceneRequestSend()** can be used to request that a scene is added to a Scene table on a remote device. This method allows the transition time for the scene to be set in units of tenths of a second (rather than seconds).
- A scene can be created by saving the current attribute settings of the relevant clusters. In this way, the current state of the system can be captured as a scene and reapplied 'at the touch of a button' when required. For example scenes can be created for lighting levels in a 'smart lighting' system. The current settings are stored as a scene in the Scene table using **eCLD_ScenesCommandStoreSceneRequestSend()**. This function can send the request to a single device or multiple devices. If a Scene table entry exists with the same scene ID and group ID, the existing cluster settings in the entry are overwritten with the new 'captured' settings.

Note: This operation of capturing the current system state as a scene does not result in meaningful settings for the transition time and scene name fields of the Scene table entry. If non-null values are required for these fields, the table entry should be created in advance with the desired field values using **eCLD_ScenesCommandAddSceneRequestSend()**.

Parent topic: Sending Remote Commands

Copying a Scene Scene settings can be copied from one scene to another scene on the same remote endpoint using the **eCLD_ScenesCommandCopySceneSceneRequestSend()** function. This function allows the settings from an existing scene with a specified source scene ID and associated group ID to be copied to a new scene with a specified destination scene ID and associated group ID.

Note: If an entry corresponding to the target scene ID and group ID exists in the Scene table on the endpoint, the entry settings are overwritten with the copied settings. Otherwise, a new Scene table entry is created with these settings.

The above function also allows all scenes associated with particular group ID to be copied to another group ID. In this case, the original scene IDs are maintained but are associated with the new group ID (any specified source and destination scene IDs are ignored). Thus, the same scene IDs are associated with two different group IDs.

Parent topic: Sending Remote Commands

Applying a Scene The cluster settings of a scene stored in the Scene table can be retrieved and applied to the system by calling **eCLD_ScenesCommandRecallSceneRequestSend()**. Again, this function can send a request to a single device or to multiple devices (using binding or group addressing).

If the required scene does not contain any settings for a particular cluster or there are some missing attribute values for a cluster, these attribute values remain unchanged in the implementation of the cluster - that is, the corresponding parts of the system do not change their states.

Parent topic: Sending Remote Commands

Deleting a Scene Two functions are provided for removing scenes from the system:

- **eCLD_ScenesCommandRemoveSceneRequestSend()** can be used to request the removal of the destination endpoint from a particular scene - that is, to remove the scene from the Scene table on the target device.
- **eCLD_ScenesCommandRemoveAllScenesRequestSend()** can be used to request that the target device removes scenes associated with a particular group ID/address - that is, remove all Scene table entries relating to this group ID. Specifying a group ID of 0x0000 removes all scenes not associated with a group.

Parent topic: Sending Remote Commands

Obtaining Information about Scenes The following functions are provided for obtaining information about scenes:

- **eCLD_ScenesCommandViewSceneRequestSend()** can be used to request information on a particular scene on the destination endpoint. Only one device may be targeted by this function. The target device returns a response containing the relevant information.
  - Alternatively, **eCLD_ScenesCommandEnhancedViewSceneRequestSend()** can be used, which allows the transition time for the scene to be obtained in units of tenths of a second (rather than seconds).
- **eCLD_ScenesCommandGetSceneMembershipRequestSend()** can be used to discover which scenes are associated with a particular group on a device. The request can be sent to a single device or to multiple devices. The target device returns a response containing the relevant information (in the case of multiple target devices, no response is returned from a device that does not contain a scene associated with the specified group ID). In this way, the function can be used to determine the unused scene IDs.

Parent topic: Sending Remote Commands

Parent topic:*Scenes Cluster*

Issuing Local Commands Some of the operations described in Section 13.4 that correspond to remote commands can also be performed locally, as described below.

Creating a Scene A scene can be created on the local node using either of the following functions:

- **eCLD_ScenesAdd():** This function can be used to add a new scene to the Scene table on the specified local endpoint. A scene ID and an associated group ID must be specified (the latter must be set to 0x0000 if there is no group association). If a scene with these IDs exists in the table, the existing entry is overwritten.
- **eCLD_ScenesStore():** This function can be used to save the currently implemented attribute values on the device to a scene in the Scene table on the specified local endpoint. A scene ID and an associated group ID must be specified (the latter must be set to 0x0000 if there is no group association). If a scene with these IDs exists in the table, the existing entry is overwritten except for the transition time and scene name fields.

Parent topic: Issuing Local Commands

Applying a Scene An existing scene can be applied on the local node using the function **eCLD_ScenesRecall()**. This function reads the stored attribute values for the specified scene from the local Scene table and implements them on the device. The values of any attributes that are not included in the scene remain unchanged.

Parent topic: Issuing Local Commands

Parent topic:*Scenes Cluster*

Functions The following Scenes cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_ScenesCreateScenes
2. eCLD_ScenesAdd
3. eCLD_ScenesStore
4. eCLD_ScenesRecall
5. eCLD_ScenesCommandAddSceneRequestSend
6. eCLD_ScenesCommandViewSceneRequestSend
7. eCLD_ScenesCommandRemoveSceneRequestSend
8. eCLD_ScenesCommandRemoveAllScenesRequestSend
9. eCLD_ScenesCommandStoreSceneRequestSend
10. eCLD_ScenesCommandRecallSceneRequestSend
11. eCLD_ScenesCommandGetSceneMembershipRequestSend
12. eCLD_ScenesCommandEnhancedAddSceneRequestSend
13. eCLD_ScenesCommandEnhancedViewSceneRequestSend
14. eCLD_ScenesCommandCopySceneSceneRequestSend

eCLD_ScenesCreateScenes

```
teZCL_Status eCLD_ScenesCreateScenes(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits,
    tsCLD_ScenesCustomDataStructure
```

(continues on next page)

(continued from previous page)

```
    *psCustomDataStructure,
tsZCL_EndPointDefinition *psEndPointDefinition);
```

Description This function creates an instance of the Scenes cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates a Scenes cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: Do not call this function for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be the first Scenes cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

On calling this function for the first time, a 'global scene' entry is created/reserved in the Scene table. On subsequent calls (for example, following a power-cycle or on waking from sleep), if the scene data is recovered by the application from non-volatile memory before the function is called then there is no reinitialization of the scene data. Note that removing all groups from the device also removes the global scene entry (along with other scene entries) from the Scene table.

The function requires an array to be declared for internal use, which contains one element (of type **uint8)** for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Scene cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Scenes cluster. This parameter can refer to a pre-filled structure called sCLD_Scenes which is provided in the **Scenes.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_Scenes which defines the attributes of Scenes cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 13.7.1)
- *psEndPointDefinition*: Pointer to the ZCL endpoint definition structure for the application (see Section 6.1.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Functions

eCLD_ScenesAdd

```
teZCL_Status eCLD_ScenesAdd(
    uint8 u8SourceEndPointId,
    uint16 u16GroupId,
    uint8 u8SceneId);
```

Description This function adds a new scene on the specified local endpoint - that is, adds an entry to the Scenes table on the endpoint. The group ID associated with the scene must also be specified (or set to 0x0000 if there is no associated group).

If a scene with the specified scene ID and group ID exists in the table, the existing entry is overwritten (that is, all previous scene data in this entry is lost).

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which Scene table entry is to be added.
- *u16GroupId*: 16-bit group ID/address of associated group (or 0x0000 if no group).
- *u8SceneId*: 8-bit scene ID of new scene.

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Functions

eCLD_ScenesStore

```
teZCL_Status eCLD_ScenesStore(
    uint8 u8SourceEndPointId,
    uint16 u16GroupId,
    uint8 u8SceneId);
```

Description This function adds a new scene on the specified local endpoint, based on the current cluster attribute values of the device- that is, saves the current attribute values of the device to a new entry of the Scenes table on the endpoint. The group ID associated with the scene must also be specified (or set to 0x0000 if there is no associated group).

If a scene with the specified scene ID and group ID exists in the table, the existing entry is overwritten. The previous scene data in this entry is lost, except for the transition time field and the scene name field - these fields are left unchanged.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which Scene table entry is to be added.
- *u16GroupId:*16-bit group ID/address of associated group (or 0x0000 if no group)
- *u8SceneId*: 8-bit scene ID of scene

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Functions

eCLD_ScenesRecall

```
teZCL_Status eCLD_ScenesRecall(
    uint8 u8SourceEndPointId,
    uint16 u16GroupId,
    uint8 u8SceneId);
```

Description This function obtains the attribute values (from the extension fields) of the scene with the specified Scene ID and Group ID on the specified (local) endpoint, and sets the corresponding cluster attributes on the device to these values. Thus, the function reads the stored attribute values for a scene and implements them on the device.

Note that the values of any cluster attributes that are not included in the scene remains unchanged.

Parameters

- *u8SourceEndPointId*: Number of local endpoint containing Scene table to be read.
- *u16GroupId*: 16-bit group ID/address of associated group (or 0x0000 if no group)
- *u8SceneId*: 8-bit scene ID of scene to be read.

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Functions

eCLD_ScenesCommandAddSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandAddSceneRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ScenesAddSceneRequestPayload *psPayload);
```

Description This function sends an Add Scene command to a remote device in order to add a scene on the specified endpoint - that is, to add an entry to the Scene table on the endpoint. The scene ID is specified in the payload of the message, along with a duration for the scene among other values (see Section 13.7.2). The scene may also be associated with a particular group.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster is registered and, if possible, add the scene to its Scene table before sending an Add Scene response indicating success or failure (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- – *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ScenesCommandViewSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandViewSceneRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ScenesViewSceneRequestPayload
*psPayload);
```

Description This function sends a View Scene command to a remote device, requesting information on a particular scene on the destination endpoint. The relevant scene ID is specified in the command payload. Note that this command can only be sent to an individual device/endpoint and not to a group address.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered and generates a View Scene response containing the relevant information (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address type eZCL_AMBOUND.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ScenesCommandRemoveSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandRemoveSceneRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ScenesRemoveSceneRequestPayload
*psPayload);
```

Description This function sends a Remove Scene command to request that the target device deletes membership of the destination endpoint from a particular scene - that is, remove the scene from the Scene table. The relevant scene ID is specified in the payload of the message. The scene may also be associated with a particular group.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered. The device then deletes the scene in the Scene table. If the request was sent to a single device (rather than to a group address), it then generates an appropriate Remove Scene response indicating success, or failure (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ScenesCommandRemoveAllScenesRequestSend

```
teZCL_Status eCLD_ScenesCommandRemoveAllScenesRequestSend(
uint8 u8SourceEndPointId,
uint8 u8DestinationEndPointId,
tsZCL_Address *psDestinationAddress,
uint8 *pu8TransactionSequenceNumber,
tsCLD_ScenesRemoveAllScenesRequestPayload
```

Description This function sends a Remove All Scenes command to request that the target device deletes all entries corresponding to the specified group ID/address in its Scene table. The relevant group ID is specified in the payload of the message. Note that specifying a group ID of 0x0000 removes all scenes not associated with a group.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered. The device then deletes the scenes in the Scene table. If the request is sent to a single device (rather than to a group address), it then generates an appropriate Remove All Scenes response indicating success, or failure (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ScenesCommandStoreSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandStoreSceneRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ScenesStoreSceneRequestPayload
```

(continues on next page)

(continued from previous page)

```
*psPayload);
```

Description This function sends a Store Scene command to request that the target device saves the current settings of all other clusters on the device as a scene - that is, adds a scene containing the current cluster settings to the Scene table. The entry is stored using the scene ID and group ID specified in the payload of the command. If an entry exists with these IDs, its existing cluster settings are overwritten with the new settings.

Note that this command does not set the transition time and scene name fields (or for a new entry, they are set to null values). If this command is to create a new scene that requires particular settings for these fields, Add Group command should be used. The scene entry must be created in advance using the Add Group command, and the fields transition time and scene name should be pre-configured.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered. If the request is sent to a single device (rather than to a group address), it then generates an appropriate Store Scene response indicating success, or failure (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: ID or number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: ID or number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_F

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ScenesCommandRecallSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandRecallSceneRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ScenesRecallSceneRequestPayload
    *psPayload);
```

Description This function sends a Recall Scene command to request that the target device retrieves and implements the settings of the specified scene - that is, reads the scene settings from the Scene table and applies them to the other clusters on the device. The required scene ID and group ID are specified in the payload of the command.

Note that if the specified scene entry does not contain any settings for a particular cluster or there are some missing attribute values for a cluster, these attribute values remains unchanged in the implementation of the cluster.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered. If the request is sent to a single device (rather than to a group address), it then generates an appropriate Recall Scene response indicating success, or failure (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE

- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ScenesCommandGetSceneMembershipRequestSend

```
teZCL_Status eCLD_ScenesCommandGetSceneMembershipRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ScenesGetSceneMembershipRequestPayload
*psPayload);
```

Description This function sends a Get Scene Membership to inquire which scenes are associated with a specified group ID on a device. The relevant group ID is specified in the payload of the command.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster is registered. If the request is sent to a single device (rather than to a group address), it then generates an appropriate Get Scene Membership response indicating success or failure and, if successful, the response contains a list of the scene IDs associated with the given group ID (see Section 13.7.3). If the original command is sent to a group address, an individual device only responds if it has scenes associated with the group ID in the command payload (so it only responds if successful).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ScenesCommandEnhancedAddSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandEnhancedAddSceneRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ScenesEnhancedAddSceneRequestPayload
*psPayload);
```

Description This function sends an Enhanced Add Scene command to a remote device in order to add a scene on the specified endpoint - that is, to add an entry to the Scene table on the endpoint. The function allows a finer transition time (in tenths of a second rather than seconds) when applying the scene. The scene ID is specified in the payload of the message, along with a duration for the scene and the transition time, among other values (see Section 13.7.2). The scene may also be associated with a particular group.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster is registered. If possible, add the scene to its Scene table before sending an Enhanced Add Scene response indicating success or failure (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.

- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ScenesCommandEnhancedViewSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandEnhancedViewSceneRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ScenesEnhancedViewSceneRequestPayload
*psPayload);
```

Description This function sends an Enhanced View Scene command to a remote device, requesting information on a particular scene on the destination endpoint. The returned information includes the finer transition time configured with the function **eCLD_ScenesCommandEnhancedAddSceneRequestSend()**. The relevant scene ID is specified in the command payload. Note that this command can only be sent to an individual device/endpoint and not to a group address.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered and generates an Enhanced View Scene response containing the relevant information (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address type eZCL_AMBOUND.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.

- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ScenesCommandCopySceneSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandCopySceneSceneRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ScenesCopySceneRequestPayload *psPayload);
```

Description This function sends a Copy Scene command to a remote device, requesting that the scene settings from one scene ID/group ID combination are copied to another scene ID/group ID combination on the target endpoint. The relevant source and destination scene ID/group ID combinations are specified in the command payload.

Note that:

- If the destination scene ID/group ID exists on the target endpoint, the existing scene is overwritten with the new settings.
- The message payload contains a 'copy all scenes' bit. If the bit is set to '1', it instructs the destination server to copy all scenes in the specified source group to scenes with the same scene IDs in the destination group. In this case, the source and destination scene IDs in the payload are ignored.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered and, if the original request is unicast, generates a Copy Scene response (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address type eZCL_AMBOUND.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

Parent topic:*Scenes Cluster*

Structures

Custom Data Structure The Scenes cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    DLIST    lScenesAllocList;
    DLIST    lScenesDeAllocList;
    tsZCL_ReceiveEventAddress        sReceiveEventAddress;
    tsZCL_CallBackEvent              sCustomCallBackEvent;
    tsCLD_ScenesCallBackMessage      sCallBackMessage;
    tsCLD_ScenesTableEntry
            asScenesTableEntry[CLD_SCENES_MAX_NUMBER_OF_SCENES];
} tsCLD_ScenesCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Custom Command Payloads The following structures contain the payloads for the Scenes cluster custom commands.

Add Scene Request Payload

```
typedef struct
{
    uint16                    u16GroupId;
    uint8                     u8SceneId;
    uint16                    u16TransitionTime;
    tsZCL_CharacterString     sSceneName;
    tsCLD_ScenesExtensionField  sExtensionField;
} tsCLD_ScenesAddSceneRequestPayload;
```

where:

- u16GroupId is the group ID with which the scene is associated (0x0000 if there is no association with a group)
- u8SceneId is the ID of the scene to be added to the Scene table (the Scene ID must be unique within the group associated with the scene)
- u16TransitionTime is the amount of time, in seconds, that the device takes to switch to this scene
- sSceneName is an optional character string (of up to 16 characters) representing the name of the scene
- sExtensionField is a structure containing the attribute values of the clusters to which the scene relates

View Scene Request Payload

```
typedef struct
{
    uint16                    u16GroupId;
    uint8                     u8SceneId;
} tsCLD_ScenesViewSceneRequestPayload;
```

where:

- u16GroupId is the group ID with which the desired scene is associated
- u8SceneId is the scene ID of the scene to be viewed

Remove Scene Request Payload

```
typedef struct
{
    uint16                    u16GroupId;
    uint8                     u8SceneId;
} tsCLD_ScenesRemoveSceneRequestPayload;
```

where:

- u16GroupId is the group ID with which the relevant scene is associated
- u8SceneId is the scene ID of the scene to be deleted from the Scene table

Remove All Scenes Request Payload

```
typedef struct
{
    uint16                    u16GroupId;
```

(continues on next page)

(continued from previous page)

```
} tsCLD_ScenesRemoveAllScenesRequestPayload;
```

where u16GroupId is the group ID for which all scenes are to be deleted.

Store Scene Request Payload

```
typedef struct
{
    uint16                    u16GroupId;
    uint8                     u8SceneId;
} tsCLD_ScenesStoreSceneRequestPayload;
```

where:

- u16GroupId is the group ID with which the relevant scene is associated
- u8SceneId is the scene ID of the scene in which the captured cluster settings are to be stored

Recall Scene Request Payload

```
typedef struct
{
    uint16                    u16GroupId;
    uint8                     u8SceneId;
} tsCLD_ScenesRecallSceneRequestPayload;
```

where:

- u16GroupId is the group ID with which the relevant scene is associated
- u8SceneId is the scene ID of the scene from which cluster settings are to be retrieved and applied

Get Scene Membership Request Payload

```
typedef struct
{
    uint16                    u16GroupId;
} tsCLD_ScenesGetSceneMembershipRequestPayload;
```

where u16GroupId is the group ID for which associated scenes are required.

Enhanced Add Scene Request Payload

```
typedef struct
{
    uint16                    u16GroupId;
    uint8                     u8SceneId;
    uint16                    u16TransitionTime100ms;
    tsZCL_CharacterString     sSceneName;
    tsCLD_ScenesExtensionField  sExtensionField;
} tsCLD_ScenesEnhancedAddSceneRequestPayload;
```

where:

- u16GroupId is the group ID with which the scene is associated (0x0000 if there is no association with a group)
- u8SceneId is the ID of the scene to be added to the Scene table (the Scene ID must be unique within the group associated with the scene)
- u16TransitionTime100ms is the amount of time, in tenths of a second, that the device takes to switch to this scene
- sSceneName is an optional character string (of up to 16 characters) representing the name of the scene
- sExtensionField is a structure containing the attribute values of the clusters to which the scene relates

View Scene Request Payload

```
typedef struct
{
    uint16                    u16GroupId;
    uint8                     u8SceneId;
} tsCLD_ScenesEnhancedViewSceneRequestPayload;
```

where:

- u16GroupId is the group ID with which the desired scene is associated
- u8SceneId is the scene ID of the scene to be viewed

Copy Scene Request Payload

```
typedef struct
{
    uint8       u8Mode;
    uint16      u16FromGroupId;
    uint8       u8FromSceneId;
    uint16      u16ToGroupId;
    uint8       u8ToSceneId;
} tsCLD_ScenesCopySceneRequestPayload;
```

where:

- u8Mode is a bitmap indicating the required copying mode (only bit 0 is used):
  - If bit 0 is set to '1', then 'copy all scenes' mode is used, in which all scenes associated with the source group are duplicated for the destination group (and the scene ID fields are ignored)
  - If bit 0 is set to '0', then a single scene is copied
- u16FromGroupId is the source group ID
- u8FromSceneId is the source scene ID (ignored for 'copy all scenes' mode)
- u16ToGroupId is the destination group ID
- u8ToSceneId is the destination scene ID (ignored for 'copy all scenes' mode)

Parent topic: Structures

Custom Command Responses The Scenes cluster generates responses to certain custom commands. The responses which contain payloads are detailed below:

Add Scene Response Payload

```
typedef struct
{
    zenum8                    eStatus;
    uint16                    u16GroupId;
    uint8                     u8SceneId;
} tsCLD_ScenesAddSceneResponsePayload;
```

where:

- eStatus is the outcome of the Add Scene command (success or invalid)
- u16GroupId is the group ID with which the added scene is associated
- u8SceneId is the scene ID of the added scene

View Scene Response Payload

```
typedef struct
{
    zenum8                    eStatus;
    uint16                    u16GroupId;
    uint8                     u8SceneId;
    uint16                    u16TransitionTime;
    tsZCL_CharacterString     sSceneName;
    tsCLD_ScenesExtensionField  sExtensionField;
} tsCLD_ScenesViewSceneResponsePayload;
```

where:

- eStatus is the outcome of the View Scene command (success or invalid)
- u16GroupId is the group ID with which the viewed scene is associated
- u8SceneId is the scene ID of the viewed scene
- u16TransitionTime is the amount of time, in seconds, that the device takes to switch to the viewed scene
- sSceneName is an optional character string (of up to 16 characters) representing the name of the viewed scene
- sExtensionField is a structure containing the attribute values of the clusters to which the viewed scene relates

Remove Scene Response Payload

```
typedef struct
{
    zenum8                    eStatus;
    uint16                    u16GroupId;
    uint8                     u8SceneId;
} tsCLD_ScenesRemoveSceneResponsePayload;
```

where:

- eStatus is the outcome of the Remove Scene command (success or invalid)
- u16GroupId is the group ID with which the removed scene is associated
- u8SceneId is the scene ID of the removed scene

Remove All Scenes Response Payload

```
typedef struct
{
    zenum8                    eStatus;
    uint16                    u16GroupId;
} tsCLD_ScenesRemoveAllScenesResponsePayload;
```

where:

- eStatus is the outcome of the Remove All Scenes command (success or invalid)
- u16GroupId is the group ID with which the removed scenes are associated

Store Scene Response Payload

```
typedef struct
{
    zenum8                    eStatus;
    uint16                    u16GroupId;
    uint8                     u8SceneId;
} tsCLD_ScenesStoreSceneResponsePayload;
```

where:

- eStatus is the outcome of the Store Scene command (success or invalid)
- u16GroupId is the group ID with which the stored scene is associated
- u8SceneId is the scene ID of the stored scene

Get Scene Membership Response Payload

```
typedef struct
{
    zenum8                    eStatus;
    uint8                     u8Capacity;
    uint16                    u16GroupId;
    uint8                     u8SceneCount;
    uint8                     *pu8SceneList;
} tsCLD_ScenesGetSceneMembershipResponsePayload;
```

where:

- eStatus is the outcome of the Get Scene Membership command (success or invalid)
- u8Capacity is the capacity of the Scene table of the device to receive more scenes - that is, the number of scenes that may be added (special values: 0xFE means that at least one more scene may be added, a higher value means that the remaining capacity of the table is unknown)
- u16GroupId is the group ID to which the query relates
- u8SceneCount is the number of scenes in the list of the next field
- pu8SceneList is a pointer to the returned list of scenes from those queried that exist on the device, where each scene is represented by its scene ID

Enhanced Add Scene Response Payload

```
typedef struct
{
    zenum8      eStatus;
    uint16      u16GroupId;
    uint8       u8SceneId;
} tsCLD_ScenesEnhancedAddSceneResponsePayload;
```

where:

- eStatus is the outcome of the Enhanced Add Scene command (success or invalid)
- u16GroupId is the group ID with which the added scene is associated
- u8SceneId is the scene ID of the added scene

Enhanced View Scene Response Payload

```
typedef struct
{
    zenum8                      eStatus;
    uint16                      u16GroupId;
    uint8                       u8SceneId;
    uint16                      u16TransitionTime;
    tsZCL_CharacterString       sSceneName;
    tsCLD_ScenesExtensionField  sExtensionField;
} tsCLD_ScenesEnhancedViewSceneResponsePayload;
```

where:

- eStatus is the outcome of the Enhanced View Scene command (success or invalid)
- u16GroupId is the group ID with which the viewed scene is associated
- u8SceneId is the scene ID of the viewed scene
- u16TransitionTime is the amount of time, in seconds, that the device takes to switch to the viewed scene
- sSceneName is an optional character string (of up to 16 characters) representing the name of the viewed scene
- sExtensionField is a structure containing the attribute values of the clusters to which the viewed scene relates

Copy Scene Response Payload

```
typedef struct
{
    uint8       u8Status;
    uint16      u16FromGroupId;
    uint8       u8FromSceneId;
} tsCLD_ScenesCopySceneResponsePayload;
```

where:

- u8Status is the outcome of the Copy Scene command (success, invalid scene, or insufficient space for new scene)
- u16FromGroupId is the source group ID for the copy
- u8FromSceneId is the source scene ID for the copy

Parent topic: Structures

Scenes Table Entry The following structure contains the data for a Scenes table entry (containing a saved scene):

```
typedef struct
{
    DNODE          dllScenesNode;
    bool           bActive;
    bool           bInTransit;
    uint16         u16GroupId;
    uint8          u8SceneId;
    uint16         u16TransitionTime;
    uint32         u32TransitionTimer;
    uint8          u8SceneNameLength;
    uint8          au8SceneName[CLD_SCENES_MAX_SCENE_NAME_LENGTH + 1];
    uint16         u16SceneDataLength;
    uint8          au8SceneData[CLD_SCENES_MAX_SCENE_STORAGE_BYTES];
#ifdef CLD_SCENES_TABLE_SUPPORT_TRANSITION_TIME_IN_MS
    uint8          u8TransitionTime100ms;
#endif
} tsCLD_ScenesTableEntry;
```

where:

- bActive is a boolean value indicating whether the scene is active (TRUE) or inactive (FALSE).
- bInTransit is a boolean value indicating whether the scene is in a transitional state (TRUE) or a constant active/inactive state (FALSE).
- u16GroupId is the identifier of the group to which the scene applies (the value 0x0000 is used to indicate that the scene is not associated with a group).
- u8SceneId is the identifier of the scene and must be a unique value within the group with which the scene is associated.
- u16TransitionTime is the length of time, in seconds, that the device takes to move from its current state to the scene state.
- u32TransitionTimer is the elapsed time, in milliseconds, since the start of a currently active transition to the scene.
- u8SceneNameLength is the number of characters in the name of the scene (and therefore the size of the array au8SceneName[] below). The value must not be greater than CLD_SCENES_MAX_SCENE_NAME_LENGTH, which is set in the compile-time options (see Section 13.9).
- au8SceneName[] is an array containing the name of the scene, with one ASCII character in each array element. The number of elements in the array must be set in u8SceneNameLength above.
- u16SceneDataLength is the number of items of data for the scene (and therefore the size of the array au8SceneData[] below). The value must not be greater than CLD_SCENES_MAX_SCENE_STORAGE_BYTES, which is set in the compile-time options (see Section 13.9).
- au8SceneData[] is an array containing the data for the scene, with one data item in each array element. The data stored is dependent on the cluster to which the scene data applies. The number of elements in the array must be set in u16SceneDataLength above.
- u8TransitionTime 100 ms is an optional that allows a fractional part to be added to the transition time (u16TransitionTime) of the scene. This value represents the number of tenths of a second in the range 0x00 to 0x09.

Parent topic: Structures

Parent topic:*Scenes Cluster*

Enumerations

teCLD_Scenes_ClusterID The following structure contains the enumerations used to identify the attributes of the Scenes cluster.

```
typedef enum
{
    E_CLD_SCENES_ATTR_ID_SCENE_COUNT          = 0x0000,   /* Mandatory */
    E_CLD_SCENES_ATTR_ID_CURRENT_SCENE,                   /* Mandatory */
    E_CLD_SCENES_ATTR_ID_CURRENT_GROUP,                   /* Mandatory */
    E_CLD_SCENES_ATTR_ID_SCENE_VALID,                     /* Mandatory */
    E_CLD_SCENES_ATTR_ID_NAME_SUPPORT,                    /* Mandatory */
    E_CLD_SCENES_ATTR_ID_LAST_CONFIGURED_BY               /* Optional  */
} teCLD_Scenes_ClusterID;
```

Parent topic: Enumerations

Parent topic:*Scenes Cluster*

Compile-time options To enable the Scenes cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_SCENES
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define SCENES_CLIENT
#define SCENES_SERVER
```

The Scenes cluster contains macros that may be optionally specified at compile time by adding some or all the following lines to the **zcl_options.h** file.

To enable the optional Last Configured By attribute, add this line:

```
#define CLD_SCENES_ATTR_LAST_CONFIGURED_BY
```

To configure the maximum length of the Scene Name storage, add this line:

```
#define CLD_SCENES_MAX_SCENE_NAME_LENGTH          (16)
```

To configure the maximum number of scenes, add this line:

```
#define CLD_SCENES_MAX_NUMBER_OF_SCENES           (16)
```

To configure the maximum number of bytes available for scene storage, add this line:

```
#define CLD_SCENES_MAX_SCENE_STORAGE_BYTES        (20)
```

To enable the Enhanced Add Scene command, add this line:

```
#define CLD_SCENES_CMD_ENHANCED_ADD_SCENE
```

To enable the Enhanced View Scene command, add this line:

```
#define CLD_SCENES_CMD_ENHANCED_VIEW_SCENE
```

To enable the Copy Scene command, add this line:

```
#define CLD_SCENES_CMD_COPY_SCENE
```

To enable TransitionTime100ms in Scene tables, add this line:

```
#define CLD_SCENES_TABLE_SUPPORT_TRANSITION_TIME_IN_MS
```

To define the value (n) of the Cluster Revision attribute, add this line:

```
#define CLD_SCENES_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Scenes Cluster*

On/Off Cluster This chapter describes the On/Off cluster.

The On/Off cluster has a Cluster ID of 0x0006.

Overview The On/Off cluster allows a device to be put into the 'on' and 'off' states, or toggled between the two states. The cluster also provides the following enhanced functionality for lighting:

- When switching off lights with an effect, saves the last light (attribute) settings to a global scene, ready to be reused for the next switch-on from the global scene - see Section 14.5.2 and Section 14.6
- Allows lights to be switched on for a timed period (and then automatically switched off) - see Section 14.5.3

To use the functionality of this cluster, you must include the file **OnOff.h** in your application and enable the cluster by defining CLD_ONOFF in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to change the on/off state of the local device.
- The cluster client is able to send commands to the server to request a change to the on/off state of the remote device.

The inclusion of the client or server software must be pre-defined in the compile-time options of the application. In addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance.

The compile-time options for the On/Off cluster are fully detailed in Section 14.10.

Parent topic:*On/Off Cluster*

On/Off Cluster Structure and Attribute The structure definition for the On/Off cluster is:

```
typedef struct
{
#ifdef ONOFF_SERVER
```

(continues on next page)

(continued from previous page)

```
    zbool                   bOnOff;
#ifdef CLD_ONOFF_ATTR_GLOBAL_SCENE_CONTROL
    zbool                   bGlobalSceneControl;
#endif
#ifdef CLD_ONOFF_ATTR_ON_TIME
    zuint16                 u16OnTime;
#endif
#ifdef CLD_ONOFF_ATTR_OFF_WAIT_TIME
    zuint16                 u16OffWaitTime;
#endif
#ifdef CLD_ONOFF_ATTR_STARTUP_ONOFF
    /* ZLO extension for OnOff Cluster    */
    zenum8                  eStartUpOnOff;
#endif
#ifdef CLD_ONOFF_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8                  u8AttributeReportingStatus;
#endif
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_OnOff;
```

where:

- bOnOff is the on/off state of the device (TRUE = on, FALSE = off).
- bGlobalSceneControl is an optional attribute for lighting that is used with the global scene - the value of this attribute determines whether to permit saving the current light settings to the global scene:
  - TRUE - Current light settings can be saved to the global scene
  - FALSE - Current light settings cannot be saved to the global scene
- u16OnTime is an optional attribute for lighting used to store the time, in tenths of a second, for which the lights remain 'on' after a switch-on with 'timed off'

  (that is, the time before starting the transition from the 'on' state to the 'off' state). The special values 0x0000 and 0xFFFF indicate that the lamp must be maintained in the 'on' state indefinitely (no timed off).
- u16OffWaitTime is an optional attribute for lighting used to store the waiting time, in tenths of a second, following a 'timed off' before the lights can be again switched on with a 'timed off'.

Note: If the bGlobalSceneControl attribute and global scene are to be used, the Scenes and Groups clusters must also be enabled - see *Chapter 13* and *Chapter 12*.

- eStartUpOnOff is an optional attribute that is used in the lighting domain to define the required start-up behavior of a light device when it is supplied with power. It determines the initial value of bOnOff on start-up. The possible values and behaviors are as follows:

eStartUpOnOff	Behavior
0x00	Put the light in the off state - set bOnOff to FALSE
0x01	Put the light in the on state - set bOnOff to TRUE
0x02	Toggle the light from its previous state:

- If bOnOff was FALSE, set it to TRUE
- If bOnOff was TRUE, set it to FALSE

| |0x03-0xFE|Reserved| |0xFF|Put the light in its previous state - set bOnOff to its previous value|

- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*On/Off Cluster*

Attributes for Default Reporting The following attribute of the On/Off cluster can be selected for default reporting:

- bOnOff

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for this attribute is described in Appendix B.3.6.

Parent topic:*On/Off Cluster*

Initialization The function **eCLD_OnOffCreateOnOff()** is used to create an instance of the On/Off cluster. The function is called by the initialization function for the host device.

Note: If the global scene is to be used to remember light settings, then. Scenes and Groups cluster instances must be created - see *Chapter 13* and *Chapter 12*.

Parent topic:*On/Off Cluster*

Sending Commands The NXP implementation of the ZCL provides functions for sending commands between an On/Off cluster client and server. A command is sent from the client to one or more endpoints on the server. Multiple endpoints can usually be targeted using binding or group addressing.

Switching On and Off A remote device (supporting the On/Off cluster server) can be switched on, switched off or, toggled between the on and off states by calling the function **eCLD_OnOffCommandSend()** on a cluster client. In the case of a toggle, if the device is initially in the on state it is switched off and if the device is initially in the off state it is switched on.

Timeout on the 'On' Command On receiving an 'On' command, a timeout is applied such that the 'on' state is maintained for a specified duration before automatically switching to the 'off' state. This timeout is defined using the optional attributes u16OnConfigurableDurationand eDurationUnitOfMeasurement. The timeout duration in seconds is given by:

u16OnConfigurableDuration * 10^(power from eDurationUnitOfMeasurement)

The attribute u16OnConfigurableDuration can be set locally or remotely, while the attribute eDurationUnitOfMeasurement must be set locally. A maximum timeout duration can be defined locally via the optional attribute u16MaxDuration, which puts an upper limit on the value of u16OnConfigurableDuration.

The attribute u16OnConfigurableDuration can be set remotely using the **eZCL_SendWriteAttributesRequest()** function. On receiving this write request, the local ZCL checks that the requested duration is within the permissible range (see Section 2.3.3.1) - if the request exceeds the maximum permitted value, the timeout duration is clipped to this maximum.

For full details of the above attributes, refer to Section 14.2.

When an 'On' command is received, an E_ZCL_CBET_CLUSTER_CUSTOM event is generated. The application is responsible for implementing the timeout described above, if it is enabled. First, the application must check the attributes u16OnConfigurableDuration and eDurationUnitOfMeasurement to make sure they have valid values. If so, the application must start a timer to implement the timeout for the duration defined by these attributes. On expiration of the timer, the application must switch from the 'on' state to the 'off' state by (locally) writing to the bOnOff attribute.

Parent topic: Switching On and Off

On/Off with Transition Effect If the Level Control cluster (see *Chapter 16*) is also used on the target device, an 'On' or 'Off' command can be implemented with a transition effect, as follows:

- If the optional Level Control 'On Transition Time' attribute is enabled, an 'On' command results in a gradual transition. This transition is from the 'off' level to the 'on' level over the time-interval specified by the attribute.
- If the optional Level Control 'Off Transition Time' attribute is enabled, an 'Off' command results in a gradual transition from the 'on' level to the 'off' level over the time-interval specified by the attribute.

Parent topic: Switching On and Off

Parent topic: Sending Commands

Switching Off Lights with Effect In the case of lighting, lights can be (remotely) switched off with an effect by calling the function **eCLD_OnOffCommandOffWithEffectSend()** on an On/Off cluster client.

Two 'off effects' are available and there are variants of each effect:

- **Fade**, with the following variants:
  - Fade to off in 0.8 seconds (default)
  - Reduce brightness by 50 % in 0.8 seconds then fade to off in 4 seconds
  - No fade
- **Rise and fall**, with (currently) only one variant:
  - Increase brightness by 20 % (if possible) in 0.5 seconds then fade to off in 1 second (default)

Parent topic: Sending Commands

Switching On Timed Lights In the case of lighting, lights can be switched on temporarily and automatically switched off at the end of a timed period. This kind of switch-on can be initiated remotely using the function **CLD_OnOffCommandOnWithTimedOffSend()** on an On/Off cluster client. In addition, a waiting time can be implemented after the automatic switch-off, during which the lights cannot be switched on again using the above function (although a normal switch-on is possible).

The following values must be specified:

- Time for which the lights remain on (in tenths of a second)
- Waiting time following the automatic switch-off (in tenths of a second)

In addition, the circumstances in which the command can be accepted must be specified - that is, accepted at any time (except during the waiting time) or only when the lights are already on. The latter case can be used to initiate a timed switch-off.

Parent topic: Sending Commands

Parent topic:*On/Off Cluster*

Saving Light Settings In the case of lighting, the current light (attribute) settings can be automatically saved to a 'global scene' when switching off the lights using the function **eCLD_OnOffCommandOffWithEffectSend()**. If the lights are, then switched on with the E_CLD_ONOFF_CMD_ON_RECALL_GLOBAL_SCENE option in **eCLD_OnOffCommandSend()**, the saved light settings are reloaded. In this way, the system remembers the last light settings used before switch-off and resumes with these settings at the next switch-on. This feature is useful when the light levels are adjustable using the Level Control cluster (*Chapter 16*) and/or the light colours are adjustable using the Colour Control cluster (*Chapter 31*).

The attribute values corresponding to the current light settings are saved (locally) to a global scene with scene ID and group ID both equal to zero. Therefore, to use this feature:

- Scenes cluster must be enabled and a cluster instance created
- Groups cluster must be enabled and a cluster instance created
- Optional On/Off cluster attribute bGlobalSceneControl must be enabled

The above attribute is a boolean which determines whether to permit the current light settings to be saved to the global scene. The attribute is set to FALSE after a switch-off using the function **eCLD_OnOffCommandOffWithEffectSend()**. It is set to TRUE after a switch-on or a change in the light settings (attributes) - more specifically, after a change resulting from a Level Control cluster 'Move to Level with On/Off' command, from a Scenes cluster 'Recall Scene' command, or from an On/Off cluster 'On' command or 'On With Recall Global Scene' command.

Parent topic:*On/Off Cluster*

Functions The following On/Off cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_OnOffCreateOnOff
2. _OnOffCommandSend
3. eCLD_OnOffCommandOffWithEffectSend
4. eCLD_OnOffCommandOnWithTimedOffSend

eCLD_OnOffCreateOnOff

```
teZCL_Status eCLD_OnOffCreateOnOff(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits,
    tsCLD_OnOffCustomDataStructure
*psCustomDataStructure);
```

Description This function creates an instance of the On/Off cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function must be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates an On/Off cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: Do not call this function for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be the first On/Off cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the On/Off cluster. The function initialises the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the On/Off cluster. This parameter can refer to a pre-filled structure called sCLD_OnOff which is provided in the **OnOff.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_OnOff which defines the attributes of On/Off cluster. The function initialises the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above)
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 14.8.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Functions

eCLD_OnOffCommandSend

```
teZCL_Status eCLD_OnOffCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    teCLD_OnOff_Command eCommand);
```

Description This function sends a custom command instructing the target device to perform the specified operation on itself: switch off, switch on, toggle (on-to-off or off-to-on), or switch on with settings retrieved from the global scene, This last option (On with Recall Global Scene) is described in Section 14.6 and, if used, must be enabled in the compile-time options on the server (target), as indicated in Section 14.10.

The device receiving this message generates a callback event on the endpoint on which the On/Off cluster was registered.

If the Level Control cluster (see *Chapter 16*) is also used on the target device, an 'On' or 'Off' command can be implemented with a transition effect, as follows:

- If the optional Level Control 'On Transition Time' attribute is enabled, an 'On' command results in a gradual transition. This transition is from the 'off' level to the 'on' level over the time-interval specified in the attribute.
- If the optional Level Control 'Off Transition Time' attribute is enabled, an 'Off' command results in a gradual transition from the 'on' level to the 'off' level over the time-interval specified in the attribute.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *eCommand*: Command code, one of the following:
  - E_CLD_ONOFF_CMD_OFF
  - E_CLD_ONOFF_CMD_ON
  - E_CLD_ONOFF_CMD_TOGGLE
  - E_CLD_ONOFF_CMD_ON_RECALL_GLOBAL_SCENE
  - E_CLD_ONOFF_CMD_TOGGLE

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_OnOffCommandOffWithEffectSend

```
teZCL_Status eCLD_OnOffCommandOffWithEffectSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_OnOff_OffWithEffectRequestPayload *psPayload);
```

Description This function sends a custom 'Off With Effect' command instructing the target lighting device to switch off one or more lights with the specified effect, which can be one of:

- fade (in two phases or no fade)
- rise and fall

Each of these effects is available in variants. The required effect and variant are specified in the command payload. For the payload details, refer to "Off With Effect Request Payload ".

The device receiving this message generates a callback event on the endpoint on which the On/Off cluster was registered.

Following a call to this function, the light settings on the target device are saved to a global scene, after which the attribute bGlobalSceneControl is set to FALSE - for more details, refer to Section 14.6.

If used, the 'Off With Effect' command must be enabled in the compile-time options on both the client and server, as described in Section 14.10.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 14.8.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_OnOffCommandOnWithTimedOffSend

```
teZCL_Status eCLD_OnOffCommandOnWithTimedOffSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_OnOff_OnWithTimedOffRequestPayload
    *psPayload);
```

Description This function sends a custom 'On With Timed Off' command instructing the target lighting device to switch on one or more lights for a timed period and then switch them off. In addition, a waiting time can be implemented after switch-off, during which the lights cannot be switched on again.

The following functionality must be specified in the command payload:

- Time for which the lights must remain on.
- Waiting time during which switched-off lights cannot be switched on again.
- Whether this command can be accepted at any time (outside the waiting time) or only when a light is on.

For the payload details, refer to "On With Timed Off Request Payload".

The device receiving this message generates a callback event on the endpoint on which the On/Off cluster was registered.

If used, the 'On With Timed Off' command must be enabled in the compile-time options on both the client and server, as described in Section 14.10.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 14.8.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

Parent topic:*On/Off Cluster*

Structures

Custom Data Structure The On/Off cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    uint8      u8Dummy;
} tsCLD_OnOffCustomDataStructure;
```

The fields are for internal use and no knowledge of them required.

Parent topic: Structures

Custom Command Payloads

Off With Effect Request Payload

```
typedef struct
{
    zuint8                  u8EffectId;
    zuint8                  u8EffectVariant;
} tsCLD_OnOff_OffWithEffectRequestPayload;
```

where:

- u8EffectId indicates the required 'off effect':
  - 0x00 - Fade
  - 0x01 - Rise and fall
    - All other values are reserved.
- u8EffectVariant indicates the required variant of the specified 'off effect' - the interpretation of this field depends on the value of u8EffectId, as indicated in the table below.

u8EffectId	u8EffectVari	Description
0x00(Fade)	0x00	Fade to off in 0.8 seconds (default)
	0x01	No fade
	0x02	Reduce brightness by 50 % in 0.8 seconds then fade to off in 4 seconds
	0x03-0xFF	Reserved
0x01(Rise and fall)	0x00	Increase brightness by 20 % (if possible) in 0.5 seconds then fade to off in 1 second (default)
	0x01-0xFF	Reserved
0x02-0xFF	0x00-0xFF	Reserved

On With Timed Off Request Payload

```
typedef struct
{
    zuint8              u8OnOff;
    zuint16             u16OnTime;
    zuint16             u16OffTime;
} tsCLD_OnOff_OnWithTimedOffRequestPayload;
```

where:

- u8OnOff indicates when the command can be accepted:
  - 0x00 - at all times (apart from in waiting time, if implemented)
  - 0x01 - only when light is on
    - All other values are reserved.
- u16OnTime is the 'on time', expressed in tenths of a second in the range 0x0000 to 0xFFFE.
- u16OffTime is the 'off waiting time', expressed in tenths of a second in the range 0x0000 to 0xFFFE

Parent topic: Structures

Parent topic:*On/Off Cluster*

Enumerations

teCLD_OnOff_ClusterID The following structure contains the enumerations used to identify the attributes of the On/Off cluster.

```
typedef enum
{
    E_CLD_ONOFF_ATTR_ID_ONOFF                       = 0x0000,
    E_CLD_ONOFF_ATTR_ID_GLOBAL_SCENE_CONTROL        = 0x4000,
    E_CLD_ONOFF_ATTR_ID_ON_TIME,
    E_CLD_ONOFF_ATTR_ID_OFF_WAIT_TIME,
    #ifdef CLD_ONOFF_ATTR_STARTUP_ONOFF
    /* ZLO extension for OnOff Cluster    */
    E_CLD_ONOFF_ATTR_ID_STARTUP_ONOFF,
    #endif
} teCLD_OnOff_ClusterID;
```

Parent topic: Enumerations

teCLD_OOSC_SwitchType (On/Off Switch Types)

```
typedef enum
{
    E_CLD_OOSC_TYPE_TOGGLE,
    E_CLD_OOSC_TYPE_MOMENTARY
} teCLD_OOSC_SwitchType;
```

Parent topic: Enumerations

teCLD_OOSC_SwitchAction (On/Off Switch Actions)

```
typedef enum
{
    E_CLD_OOSC_ACTION_S2ON_S1OFF,
    E_CLD_OOSC_ACTION_S2OFF_S1ON,
    E_CLD_OOSC_ACTION_TOGGLE
} teCLD_OOSC_SwitchAction;
```

Parent topic: Enumerations

Parent topic:*On/Off Cluster*

Compile-time options

- To enable the On/Off cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_ONOFF
```

- In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define ONOFF_CLIENT
#define ONOFF_SERVER
```

- The On/Off cluster contains macros that may be optionally specified at compile time by adding some or all of the following lines to the **zcl_options.h** file.

Optional Attributes

1. To enable the optional On Configurable Duration attribute, add this line:

```
#define CLD_ONOFF_ATTR_ID_ON_CONFIGURABLE_DURATION
```

2. To enable the optional Duration Unit of Measure attribute, add this line:

```
#define CLD_ONOFF_ATTR_ID_DURATION_UNIT_OF_MEASUREMENT
```

3. To enable the optional Maximum Duration attribute, add this line:

```
#define CLD_ONOFF_ATTR_ID_MAX_DURATION
```

4. To enable the optional Global Scene Control attribute, add this line:

```
#define CLD_ONOFF_ATTR_GLOBAL_SCENE_CONTROL
```

5. To enable the optional On Time attribute, add this line:

```
#define CLD_ONOFF_ATTR_ON_TIME
```

6. To enable the optional Off Wait Time attribute, add this line:

```
#define CLD_ONOFF_ATTR_OFF_WAIT_TIME
```

7. To enable the optional Start-up On/Off attribute, add this line:

```
#define CLD_ONOFF_ATTR_STARTUP_ONOFF
```

8. To enable the optional Attribute Reporting Status attribute, add this line:

```
#define CLD_ONOFF_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Optional Commands

1. Add this line to enable processing of the On With Recall Global Scene command on the server:

```
#define CLD_ONOFF_CMD_ON_WITH_RECALL_GLOBAL_SCENE
```

2. Add this line to enable the optional On With Timed Off command on the client and server:

```
#define CLD_ONOFF_CMD_ON_WITH_TIMED_OFF
```

3. Add this line to enable the optional Off With Effect command on the client and server:

```
#define CLD_ONOFF_CMD_OFF_WITH_EFFECT
```

Cluster Revision To define the value (n) of the Cluster Revision attribute, add this line:

```
#define CLD_ONOFF_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*On/Off Cluster*

On/Off Switch Configuration Cluster This chapter describes the On/Off Switch Configuration cluster.

The On/Off Switch Configuration cluster has a Cluster ID of 0x0007.

Note: When using this cluster, the On/Off cluster must also be used (see *Chapter 14*).

Overview The On/Off Switch Configuration cluster allows the switch type on a device to be defined, as well as the commands to be generated when the switch is moved between its two states.

To use the functionality of this cluster, you must include the file **OOSC.h** in your application and enable the cluster by defining CLD_OOSC in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to define a switch configuration.
- The cluster client is able to send commands to define a switch configuration.

The inclusion of the client or server software must be pre-defined in the compile-time options of the application. In addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance.

The compile-time options for the On/Off Switch Configuration cluster are fully detailed in Section 15.6.

Parent topic:*On/Off Switch Configuration Cluster*

On/Off Switch Config Cluster Structure and Attribute The structure definition for the On/Off Switch Configuration cluster is:

```
typedef struct
{
#ifdef OOSC_SERVER
    zenum8              eSwitchType;
    zenum8              eSwitchActions;
#endif
    zuint16             u16ClusterRevision;
} tsCLD_OOSC;
```

where:

- eSwitchType is the type of the switch, one of:
  - Toggle (0x00) - when the switch is physically moved between its two states, it remains in the latest state until it is physically returned to the original state (for example, a rocker switch)
  - Momentary (0x01) - when the switch is physically moved between its two states, it returns to the original state as soon as it is released (for example, a pushbutton which is pressed and then released)
  - Multi-function (0x02) - when the switch is physically moved between its two states, the command it sends is application-specific and may be dependent on the circumstances.
    - Enumerations are provided for the above settings (see Section 15.5.2).
- eSwitchActions defines the commands to be generated when the switch moves between state 1 (S1) and state 2 (S2), one of:
  - S1 to S2 is 'switch on', S2 to S1 is 'switch off'
  - S1 to S2 is 'switch off', S2 to S1 is 'switch on'
  - S1 to S2 is 'toggle', S2 to S1 is 'toggle'
    - Enumerations are provided for the above settings (see Section 15.5.3).
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section2.4.

Parent topic:*On/Off Switch Configuration Cluster*

Initialisation The function **eCLD_OOSCCreateOnOffSwitchConfig()** is used to create an instance of the On/Off Switch Configuration cluster. The function is called by the initialization function for the host device.

Parent topic:*On/Off Switch Configuration Cluster*

Functions The following On/Off Switch Configuration cluster function is provided in the NXP implementation of the ZCL:

- eCLD_OOSCCreateOnOffSwitchConfig

eCLD_OOSCCreateOnOffSwitchConfig

```
teZCL_Status eCLD_OOSCCreateOnOffSwitchConfig(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    tsZCL_AttributeStatus *psAttributeStatus);
```

Description This function creates an instance of the On/Off Switch Configuration cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

Call the function when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates an On/Off Switch Configuration cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be the first On/Off Switch Configuration cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:

  TRUE - server

  FALSE - client
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the On/Off Switch Configuration cluster. This parameter can refer to a pre-filled structure called sCLD_OOSC which is provided in the **OOSC.h** file.

- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_OOSC which defines the attributes of On/Off Switch Configuration cluster. The function initializes the attributes with default values.
- *psAttributeStatus* Pointer to a structure containing the storage for each attribute's status

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Functions

Parent topic:*On/Off Switch Configuration Cluster*

Enumerations

teCLD_OOSC_ClusterID The following structure contains the enumerations used to identify the attributes of the On/Off Switch Configuration cluster.

```
typedef enum
{
    E_CLD_OOSC_ATTR_ID_SWITCH_TYPE       = 0x0000,   /* Mandatory */
    E_CLD_OOSC_ATTR_ID_SWITCH_ACTIONS    = 0x0010,   /* Mandatory */
} teCLD_OOSC_ClusterID;
```

Parent topic: Enumerations

teCLD_OOSC_SwitchType The following structure contains the enumerations used to specify the switch type in the eSwitchType attribute.

```
typedef enum
{
    E_CLD_OOSC_TYPE_TOGGLE,
    E_CLD_OOSC_TYPE_MOMENTARY,
    E_CLD_OOSC_TYPE_MULTI_FUNCTION
} teCLD_OOSC_SwitchType;
```

The above enumerations are detailed in the table below.

Enumeration	Description
E_CLD_OOSC_TY	Toggle - when the switch is physically moved between its two states, it remains in the latest state until it is physically returned to the original state (for example, a rocker switch).
E_CLD_OOSC_TY	Momentary - when the switch is physically moved between its two states, it returns to the original state as soon as it is released (for example, a pushbutton which is pressed and then released).
E_CLD_OOSC_TY	Multi-function - when the switch is physically moved between its two states, the command it sends is application-specific and may be dependent on the circumstances.

Parent topic: Enumerations

teCLD_OOSC_SwitchAction The following structure contains the enumerations used to specify the switch action in the eSwitchActions attribute.

```
typedef enum
{
    E_CLD_OOSC_ACTION_S2ON_S1OFF,
    E_CLD_OOSC_ACTION_S2OFF_S1ON,
    E_CLD_OOSC_ACTION_TOGGLE
} teCLD_OOSC_SwitchAction;
```

The above enumerations are detailed in the .

|**Enumeration**|**Description**

When the switch moves between state 1 (S1) and state 2 (S2)...

	———————	———————————————————————————————————
E_CLD_OOSC_ACTION_S2ON_S1OFF	S1 to S2 is 'switch on', S2 to S1 is 'switch off'
E_CLD_OOSC_ACTION_S2OFF_S1ON	S1 to S2 is 'switch off', S2 to S1 is 'switch on'
E_CLD_OOSC_ACTION_TOGGLE	S1 to S2 is 'toggle', S2 to S1 is 'toggle'

Parent topic: Enumerations

Parent topic:*On/Off Switch Configuration Cluster*

Compile-time options To enable the On/Off Switch Configuration cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_OOSC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define OOSC_CLIENT
#define OOSC_SERVER
```

To define the value (n) of the Cluster Revision attribute, add this line:

```
#define CLD_OOSC_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*On/Off Switch Configuration Cluster*

Level Control Cluster This chapter describes the Level Control cluster.

The Level Control cluster has a Cluster ID of 0x0008.

Overview The Level Control cluster is used to control the level of a physical quantity on a device. The physical quantity is device-dependent - for example, it could be light, sound or heat output.

Note: This cluster should be used with the On/Off cluster (see *Chapter 14*) and this is assumed to be the case in this description.

The Level Control cluster provides the facility to increase to a target level gradually during a 'switch-on' and decrease from this level gradually during a 'switch-off'.

To use the functionality of this cluster, you must include the file **LevelControl.h** in your application and enable the cluster by defining CLD_LEVEL_CONTROL in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to change the level on the local device.
- The cluster client is able to send commands to change the level on the remote device.

The inclusion of the client or server software must be pre-defined in the application's compile-time options. In addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance.

The compile-time options for the Level Control cluster are fully detailed in Section 16.10.

Parent topic:*Level Control Cluster*

Level Control Cluster structure and attributes The structure definition for the Level Control cluster is shown below.

```
typedef struct
{
#ifdef LEVEL_CONTROL_SERVER
    zuint8              u8CurrentLevel;
#ifdef CLD_LEVELCONTROL_ATTR_REMAINING_TIME
    zuint16             u16RemainingTime;
#endif
    zbmap8              u8Options;
#ifdef CLD_LEVELCONTROL_ATTR_ON_OFF_TRANSITION_TIME
    zuint16             u16OnOffTransitionTime;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_ON_LEVEL
    zuint8              u8OnLevel;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_ON_TRANSITION_TIME
    zuint16             u16OnTransitionTime;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_OFF_TRANSITION_TIME
    zuint16             u16OffTransitionTime;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_DEFAULT_MOVE_RATE
    zuint8              u8DefaultMoveRate;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_STARTUP_CURRENT_LEVEL
    zuint8              u8StartUpCurrentLevel;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8              u8AttributeReportingStatus;
#endif
#endif
    zuint16             u16ClusterRevision;
} tsCLD_LevelControl;
```

where:

- u8CurrentLevel is the current level on the device, in the range 0x01 to 0xFE (0x00 is not used and 0xFF represents an undefined level).
- u16RemainingTime is the time remaining (in tenths of a second) at the current level
- u8Options is a bitmap which allows behaviors connected with certain commands to be defined (these behaviors should only be defined during commissioning), as follows:

Bits	Name	Description
0	Chang	Defines whether changes to the Level Control cluster can be made from control clusters (for example, Colour Control) when the bOnOff attribute of the On/Off cluster is zero (off):

- 1 – Allow changes
- 0 – Do not allow changes

| |1|CoupleColorTempToLevel|Defines whether changes to the u8CurrentLevel attribute are to be coupled with colour temperature:

- 1 – Couple changes
- 0 – Do not couple changes

| |2-7|-|Reserved|

- u16OnOffTransitionTime is the time taken (in tenths of a second) to increase from 'off' to the target level or decrease from the target level to 'off' when an On or Off command is received, respectively (see below for target level)
- u8OnLevel is the target level to which u8CurrentLevel is set when an On command is received. The value must be in the range 0x01 to 0xFE. If maximum and minimum levels are implemented using the final four attributes of the cluster (see below), the value must be within the permissible range.
- u16OnTransitionTime is an optional attribute representing the time taken (in tenths of a second) to increase the level from 0 (off) to 255 (on) when an 'On' command of the On/Off cluster is received. The special value of 0xFFFF indicates that the transition time u16OnOffTransitionTime must be used instead (which is also used if u16OnTransitionTime is not enabled).
- u16OffTransitionTime is an optional attribute representing the time taken (in tenths of a second) to decrease the level from 255 (on) to 0 (off) when an 'Off' command of the On/Off cluster is received. The special value of 0xFFFF indicates that the transition time u16OnOffTransitionTime must be used instead (which is also be used if u16OffTransitionTime is not enabled).
- u8DefaultMoveRate is an optional attribute representing the rate of movement (in units per second) to be used when a Move command is received with a rate value (u8Rate) equal to 0xFF (see Section 16.8.3.2).
- u8StartUpCurrentLevel is an optional attribute that is used in the lighting domain to define the required start-up level of a light device when it is supplied with power. It determines the initial value of u8CurrentLevel on start-up (in the range 0x01 to 0xFE).
- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Level Control Cluster*

Attributes for Default Reporting The following attribute of the Level Control cluster can be selected for default reporting:

- u8CurrentLevel

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for this attribute is described in Appendix B.3.6.

Parent topic:*Level Control Cluster*

Initialization The function **eCLD_LevelControlCreateLevelControl()** is used to create an instance of the Level Control cluster. The function is called by the initialization function for the host device.

Parent topic:*Level Control Cluster*

Sending Remote Commands The NXP implementation of the ZCL provides functions for sending commands between a Level Control cluster client and server. A command is sent from the client to one or more endpoints on the server. Multiple endpoints can usually be targeted using binding or group addressing.

Changing Level Three functions (see below) are provided for sending commands to change the current level on a device. These commands modify the 'current level' attribute of the Level Control cluster.

Each of the three level functions can be implemented with the On/Off cluster. In this case:

- If the command increases the current level, the OnOff attribute of the On/Off cluster is set to 'on'.
- If the command decreases the current level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to 'off'.

Use of the three functions/commands are described below.

Move to Level Command The current level can be moved (up or down) to a new level over a given time using the function **eCLD_LevelControlCommandMoveToLevelCommandSend()**. The target level and transition time are specified in the command payload (see Section16.8.3.1).

Move Command The current level can be moved (up or down) at a specified rate using the function **eCLD_LevelControlCommandMoveCommandSend()**. The level varies until stopped (see Section 16.5.2) or until the maximum or minimum level is reached. The direction and rate are specified in the command payload (see Section 16.8.3.2).

Step Command The current level can be moved (up or down) to a new level in a single step over a given time using the function **eCLD_LevelControlCommandStepCommandSend()**. The direction, step size and, transition time are specified in the command payload (see Section 16.8.3.3).

Parent topic: Sending Remote Commands

Stopping a Level Change A level change initiated using any of the functions referenced in Section 16.5.1 can be halted using the function **eCLD_LevelControlCommandStopCommandSend()** or**eCLD_LevelControlCommandStopWithOnOffCommandSend()**.

Parent topic: Sending Remote Commands

Parent topic:*Level Control Cluster*

Issuing Local Commands Some of the operations described in Section 16.5 that correspond to remote commands can also be performed locally, as described below.

Setting Level The level on the device on a local endpoint can be set using the function **eCLD_LevelControlSetLevel()**. This function sets the value of the 'current level' attribute of the Level Control cluster. A transition time must also be specified, in units of tenths of a second, during which the level moves toward the target value (this transition should be as smooth as possible, not stepped).

The specified level must be in the range 0x01 to 0xFE (the extreme values 0x00 and 0xFF are not used), where:

- 0x01 represents the minimum possible level for the device
- 0x02 to 0xFD are device-dependent values
- 0xFE represents the maximum level for the device

When the On/Off cluster is also enabled, calling the above function can have the following outcomes:

- If the operation is to increase the current level, the OnOff attribute of the On/Off cluster is set to 'on'.
- If the operation is to decrease the current level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to 'off'.

Parent topic: Issuing Local Commands

Obtaining Level The current level on the device on a local endpoint can be obtained using the function **eCLD_LevelControlGetLevel()**. This function reads the value of the 'current level' attribute of the Level Control cluster.

Parent topic: Issuing Local Commands

Parent topic:*Level Control Cluster*

Functions The following Level Control cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_LevelControlCreateLevelControl
2. eCLD_LevelControlSetLevel
3. eCLD_LevelControlGetLevel
4. eCLD_LevelControlCommandMoveToLevelCommandSend
5. eCLD_LevelControlCommandMoveCommandSend
6. eCLD_LevelControlCommandStepCommandSend
7. eCLD_LevelControlCommandStopCommandSend

eCLD_LevelControlCreateLevelControl

```
teZCL_Status eCLD_LevelControlCreateLevelControl(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits,
    tsCLD_LevelControlCustomDataStructure
```

(continues on next page)

(continued from previous page)

```
*psCustomDataStructure);
```

Description This function creates an instance of the Level Control cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

Call the function when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates a Level Control cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be the first Level Control cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Level Control cluster. The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Level Control cluster. This parameter can refer to a pre-filled structure called sCLD_LevelControl which is provided in the **LevelControl.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_LevelControl, which defines the attributes of Level Control cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above)
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 16.8.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Functions

eCLD_LevelControlSetLevel

```
teZCL_Status eCLD_LevelControlSetLevel(
    uint8 u8SourceEndPointId,
    uint8 u8Level,
    uint16 u16TransitionTime);
```

Description This function sets the level on the device on the specified (local) endpoint by writing the specified value to the 'current level' attribute. The new level is implemented over the specified transition time by gradually changing the level.

The specified target level must be within the range 0x01 to 0xFE or a more restricted range imposed by the device manufacturer and/or user/installer (see Section 16.6.1).

This operation is performed with the On/Off cluster (if enabled), in which case:

- If the operation is to increase the current level, the OnOff attribute of the On/Off cluster is set to 'on'.
- If the operation is to decrease the current level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to 'off'.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint on which level is to be changed
- *u8Level*: New level to be set, within the range 0x01 to 0xFE or within a more restricted range (see above)
- *u16TransitionTime*: Time to be taken, in units of tenths of a second, to reach the target level (0xFFFF means move to the level as fast as possible)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_LevelControlGetLevel

```
teZCL_Status eCLD_LevelControlGetLevel(
    uint8 u8SourceEndPointId,
    uint8 *pu8Level);
```

Description This function obtains the current level on the device on the specified (local) endpoint by reading the 'current level' attribute.

Parameters

- *u8SourceEndPointId* Number of the local endpoint from which the level is to be read
- *pu8Level* Pointer to location to receive obtained level

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_LevelControlCommandMoveToLevelCommandSend

```
eCLD_LevelControlCommandMoveToLevelCommandSend
    teZCL_Status eCLD_LevelControlCommandMoveToLevelCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    bool_t bWithOnOff,
    tsCLD_LevelControl_MoveToLevelCommandPayload
    *psPayload);
```

Description This function sends a Move to Level command to instruct a device to move its 'current level' attribute to the specified level over a specified time. The new level and the transition time are specified in the payload of the command (see Section 16.8.3). The target level must be within the range 0x01 to 0xFE or a more restricted range imposed by the device manufacturer and/or user/installer (see Section 16.5.1).

The device receiving this message generates a callback event on the endpoint on which the Level Control cluster is registered and transition the 'current level' attribute to the new value.

The option is provided to use this command in association with the On/Off cluster. In this case:

- If the command is to increase the current level, the OnOff attribute of the On/Off cluster is set to 'on'.
- If the command is to decrease the current level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to 'off'.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *bWithOnOff*: Specifies whether this cluster interacts with the On/Off cluster:
  - TRUE - interaction
  - FALSE - no interaction
- *psPayload*Pointer to a structure containing the payload for this message (see Section 16.8.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_LevelControlCommandMoveCommandSend

```
teZCL_Status eCLD_LevelControlCommandMoveCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    bool_t bWithOnOff,
    tsCLD_LevelControl_MoveCommandPayload
    *psPayload);
```

Description This function sends a Move command to instruct a device to move its 'current level' attribute either up or down in a continuous manner at a specified rate. The direction and rate are specified in the payload of the command (see Section 16.8.3).

If the current level reaches the maximum or minimum permissible level for the device, the level change stops.

The device receiving this message generates a callback event on the endpoint on which the Level Control cluster is registered, and move the current level in the direction and at the rate specified.

The option is provided to use this command in association with the On/Off cluster. In this case:

- If the command is to increase the current level, the OnOff attribute of the On/Off cluster is set to 'on'.
- If the command decreases the current level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to 'off'.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *bWithOnOff*: Specifies whether this cluster interacts with the On/Off cluster:
  - TRUE - interaction
  - FALSE - no interaction
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 16.8.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_LevelControlCommandStepCommandSend

```
teZCL_Status eCLD_LevelControlCommandStepCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
```

(continues on next page)

(continued from previous page)

```
bool_t bWithOnOff,
tsCLD_LevelControl_StepCommandPayload
*psPayload);
```

Description This function sends a Step command to instruct a device to move its 'current level' attribute either up or down in a step of the specified step size over the specified time. The direction, step size and, transition time are specified in the payload of the command (see Section 16.8.3).

If the target level is above the maximum or below the minimum permissible level for the device, the stepped change is limited to this level, and the transition time is cut short.

The device receiving this message generates a callback event on the endpoint on which the Level Control cluster is registered and move the current level according to the specified direction, step size and transition time.

The option is provided to use this command in association with the On/Off cluster. In this case:

- If the command is to increase the current level, the OnOff attribute of the On/Off cluster is set to 'on'.
- If the command decreases the current level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to 'off'.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *bWithOnOff*: Specifies whether this cluster interacts with the On/Off cluster:
  - TRUE - interaction
  - FALSE - no interaction
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 16.8.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE

- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_LevelControlCommandStopCommandSend

```
teZCL_Status eCLD_LevelControlCommandStopCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    bool_t bWithOnOff,
    tsCLD_LevelControl_StopCommandPayload
*psPayload);*
```

Description This function sends a Stop command to instruct a device to halt any transition to a new level. If necessary, the command can be sent as the 'with On/Off' version, used when the Level Control cluster interacts with the On/Off cluster, but the result on the target device is the same.

The device receiving this message generates a callback event on the endpoint on which the Level Control cluster is registered and stop any in progress transition.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *bWithOnOff* Specifies whether this cluster interacts with the On/Off cluster:

  TRUE - interaction

  FALSE - no interaction
- *psPayload* Pointer to a structure containing the payload for this message (see Section 16.8.3.4)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

Parent topic:*Level Control Cluster*

Structures

Level Control Transition Structure The following structure is used to store information about an ongoing level transition.

```
typedef struct
{
    teCLD_LevelControl_Transition    eTransition;
    teCLD_LevelControl_MoveMode      eMode;
    bool bWithOnOff;
    int  iCurrentLevel;
    int  iTargetLevel;
    int  iPreviousLevel;
    int  iStepSize;
    uint32      u32Time;
} tsCLD_LevelControl_Transition;
```

where:

- eTransition is an enumeration indicating the type of level transition implemented - for the enumerations, see Section 16.9.2.
- eMode is an enumeration indicating the direction in which the level is moved during the transition - for the enumerations, see Section 16.9.3
- bWithOnOff is a boolean which is set to TRUE if the transition is implemented with the On/Off cluster (or FALSE otherwise). When enabled:
  - If the transition is to increase the level, the OnOff attribute of the On/Off cluster is set to ‘on’.
  - If the transition decreases the level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to ‘off’.
- iCurrentLevel is the current level (0x01-0xFE) during the transition.
- iTargetLevel is the target level (0x01-0xFE) of the transition.
- iPreviousLevel is the previous level (0x01-0xFE) during the transition.
- iStepSize is the size of a single step of the transition.
- u32Time is the total time for the transition, in tenths of a second.

Parent topic: Structures

Custom Data Structure The Level Control cluster requires extra storage space to be allocated for use by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsCLD_LevelControl_Transition        sTransition;
    tsZCL_ReceiveEventAddress            sReceiveEventAddress;
    tsZCL_CallBackEvent                  sCustomCallBackEvent;
    tsCLD_LevelControlCallBackMessage    sCallBackMessage;
} tsCLD_LevelControlCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Custom Command Payloads The following structures contain the payloads for the Level Control cluster custom commands.

Move To Level Command Payload

```
typedef struct
{
    uint8                u8Level;
    uint16               u16TransitionTime;
    zbmap8               u8OptionsMask;
    zbmap8               u8OptionsOverride;
} tsCLD_LevelControl_MoveToLevelCommandPayload;
```

where:

- u8Level is the target level within the range 0x01 to 0xFE or within a more restricted range (see Section 16.5.1)
- u16TransitionTime is the time taken, in units of tenths of a second, to reach the target level (0xFFFF means use the u16OnOffTransitionTime attribute instead - if this optional attribute is not present, the device changes the level as fast as possible).
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Parent topic: Custom Command Payloads

Move Command Payload

```
typedef struct
{
    uint8                u8MoveMode;
    uint8                u8Rate;
    zbmap8               u8OptionsMask;
    zbmap8               u8OptionsOverride;
} tsCLD_LevelControl_MoveCommandPayload;
```

where:

- u8MoveMode indicates the direction of the required level change, up (0x00) or down (0x01)
- u8Rate represents the required rate of change in units per second (0xFF means use the u``8DefaultMoveRate attribute instead - if this optional attribute is not present, the device changes the level as fast as possible)
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Parent topic: Custom Command Payloads

Step Command Payload

```
typedef struct
{
    uint8                u8StepMode;
    uint8                u8StepSize;
    uint16               u16TransitionTime;
    zbmap8               u8OptionsMask;
    zbmap8               u8OptionsOverride;
} tsCLD_LevelControl_StepCommandPayload;
```

where:

- u8StepMode indicates the direction of the required level change, up (0x00) or down (0x01)
- u8StepSize is the size for the required level change
- u16TransitionTime is the time taken, in units of tenths of a second, to reach the target level (0xFFFF means move to the level as fast as possible)
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Parent topic: Custom Command Payloads

Stop Command Payload

```
typedef struct
{
    zbmap8               u8OptionsMask;
    zbmap8               u8OptionsOverride;
} tsCLD_LevelControl_StopCommandPayload;
```

where OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Parent topic: Custom Command Payloads

Parent topic: Structures

Parent topic:*Level Control Cluster*

Enumerations

teCLD_LevelControl_ClusterID The following enumerations are used to identify the attributes of the Level Control cluster.

```
typedef enum
{
    E_CLD_LEVELCONTROL_ATTR_ID_CURRENT_LEVEL                = 0x0000,
    E_CLD_LEVELCONTROL_ATTR_ID_REMAINING_TIME,
    E_CLD_LEVELCONTROL_ATTR_ID_OPTIONS                      = 0x000F,
    E_CLD_LEVELCONTROL_ATTR_ID_ON_OFF_TRANSITION_TIME       = 0x010,
    E_CLD_LEVELCONTROL_ATTR_ID_ON_LEVEL,
    E_CLD_LEVELCONTROL_ATTR_ID_ON_TRANSITION_TIME,
    E_CLD_LEVELCONTROL_ATTR_ID_OFF_TRANSITION_TIME,
    E_CLD_LEVELCONTROL_ATTR_ID_DEFAULT_MOVE_RATE,
    E_CLD_LEVELCONTROL_ATTR_ID_STARTUP_CURRENT_LEVEL        = 0x4000,
} teCLD_LevelControl_ClusterID;
```

Parent topic: Enumerations

teCLD_LevelControl_Transition

```
The following enumerations are used to specify a type of level transition.
typedef enum
{
    E_CLD_LEVELCONTROL_TRANSITION_MOVE_TO_LEVEL = 0,
    E_CLD_LEVELCONTROL_TRANSITION_MOVE,
    E_CLD_LEVELCONTROL_TRANSITION_STEP,
    E_CLD_LEVELCONTROL_TRANSITION_STOP,
    E_CLD_LEVELCONTROL_TRANSITION_ON,
    E_CLD_LEVELCONTROL_TRANSITION_OFF,
    E_CLD_LEVELCONTROL_TRANSITION_OFF_WITH_EFFECT_DIM_DOWN_FADE_OFF,
    E_CLD_LEVELCONTROL_TRANSITION_OFF_WITH_EFFECT_DIM_UP_FADE_OFF,
    E_CLD_LEVELCONTROL_TRANSITION_NONE,
} teCLD_LevelControl_Transition;
```

Parent topic: Enumerations

teCLD_LevelControl_MoveMode

```
The following enumerations are used to specify the direction of a level change.
typedef enum
{
    E_CLD_LEVELCONTROL_MOVE_MODE_UP = 0x0,
    E_CLD_LEVELCONTROL_MOVE_MODE_DOWN
} teCLD_LevelControl_MoveMode;
```

Parent topic: Enumerations

Parent topic:*Level Control Cluster*

Compile-time options To enable the Level Control cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_LEVEL_CONTROL
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define LEVEL_CONTROL_CLIENT
#define LEVEL_CONTROL_SERVER
```

The Level Control cluster contains macros that may be optionally specified at compile time by adding one or more of the following lines to the **zcl_options.h** file.

Optional Attributes To enable the optional Remaining Time attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_REMAINING_TIME
```

To enable the optional On/Off Transition Time attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_ON_OFF_TRANSITION_TIME
```

To enable the optional On Level attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_ON_LEVEL
```

To enable the optional On Transition Time attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_ON_TRANSITION_TIME
```

To enable the optional Off Transition Time attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_OFF_TRANSITION_TIME
```

To enable the optional Default Move Rate attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_DEFAULT_MOVE_RATE
```

To enable the optional Start-up Current Level attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_STARTUP_CURRENT_LEVEL
```

Global Attributes To enable the optional Attribute Reporting Status attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

To define the value (n) of the Cluster Revision attribute, add this line:

```
#define CLD_LEVELCONTROL_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Level Control Cluster*

Alarms Cluster This chapter describes the Alarms cluster which is defined in the ZCL.

The Alarms cluster has a Cluster ID of 0x0009.

Overview The Alarms cluster is used to configure alarm functionality on a device and send alarm notifications to other devices.

Note: The Alarms cluster is used with other clusters that use alarms. Alarms conditions and codes are cluster-specific and defined in these clusters.

To use the functionality of this cluster, you must include the file **Alarms.h** in your application and enable the cluster by defining CLD_ALARMS in the **zcl_options.h** file.

An Alarms cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Alarms cluster are fully detailed in Section 17.9.

The Alarms cluster server resides on a device on which other clusters may generate alarm conditions (for example, a cluster attribute value exceeds a certain limit). When an alarm condition occurs, the Alarms cluster server may send an Alarm notification to a cluster client - for example, the client may be on a device that signals alarms to the user. An Alarms cluster client may also contain a user interface (for example, a set of buttons) which allows user instructions to be sent to the server - for example, to reset an alarm.

The Alarms cluster server implements alarm logging by keeping a record of the previously generated alarms in an Alarms table. Thus, historic alarm information can be retrieved from the Alarms table. Each entry of the table contains the following information about one alarm activation:

- Alarm code which identifies the type of alarm (this type is cluster-specific)
- Cluster ID of the cluster which generated the alarm
- Time-stamp indicating the time (UTC) at which the alarm is generated

A maximum number of Alarms table entries can be set in the compile-time options.

Note: Any device which implements time-stamping for alarms must also employ the Time cluster, described in *Chapter 17*.

Parent topic:*Alarms Cluster*

Alarms Cluster structure and attributes The structure definition for the Alarms cluster is shown below.

```
typedef struct
{
#ifdef ALARMS_SERVER
    #ifdef CLD_ALARMS_ATTR_ALARM_COUNT
        zuint16             u16AlarmCount;
    #endif
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_Alarms;
```

where:

- u``16AlarmCount is an optional attribute which contains the number of entries currently in the Alarms table on the cluster server.

- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Alarms Cluster*

Initialization The function **eCLD_AlarmsCreateAlarms()** is used to create an instance of the Alarms cluster. The function is generally called by the initialization function for the host device.

Parent topic:*Alarms Cluster*

Alarm Operations This section describes the main operations that are performed using the Alarms cluster - raising an alarm and clearing/resetting an alarm.

Raising an Alarm An alarm is raised when an alarm condition occurs on a cluster on the same endpoint as the Alarms cluster server - for example, when a cluster attribute falls below a lower threshold. The Alarms cluster server should then send an Alarm notification to any remote Alarms cluster clients that might be interested in the alarm. The server application can send this notification and add an entry to the Alarms table by calling the **eCLD_AlarmsSignalAlarm()** function. On arriving at a destination device, the notification causes an E_CLD_ALARMS_CMD_ALARM event to be generated to notify the client application.

Parent topic: Alarm Operations

Resetting Alarms (from Client) A client application can remotely request one alarm or all alarms to be reset:

- The function **eCLD_AlarmsCommandResetAlarmCommandSend()** can be used to request an individual alarm to be reset. A Reset Alarm command is sent to the cluster server. On arriving at the destination device, the command causes an E_CLD_ALARMS_CMD_RESET_ALARM event to be generated.
- The function **eCLD_AlarmsCommandResetAllAlarmsCommandSend()** can be used to request all alarms to be reset. A Reset All Alarms command is sent to the cluster server. On arriving at the destination device, the command causes an E_CLD_ALARMS_CMD_RESET_ALL_ALARMS event to be generated.

On the generation of the above events on the cluster server, the server application can remove the relevant entry or entries from the local Alarms table as described in *Section 17.4.2*.

Note: The client application can also request that all the entries in an Alarms table are removed by calling **eCLD_AlarmsCommandResetAlarmLogCommandSend()**. In this case, the entries are automatically deleted by the ZCL on the server.

Parent topic: Alarm Operations

Parent topic:*Alarms Cluster*

Alarms Events The Alarms cluster has its own events that are handled through the callback mechanism outlined in *Chapter 3*. If a device uses the Alarms cluster, then Alarms event handling must be included in the callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function (for example, through **eHA_RegisterThermostatEndPoint()** for a Thermostat device). The relevant callback function is then invoked when an Alarms event occurs.

For an Alarms event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_AlarmsCallBackMessage structure:

```
typedef struct
{
 uint8  u8CommandId;
 union
  {
 tsCLD_AlarmsResetAlarmCommandPayload  *psResetAlarmCommandPayload;
  tsCLD_AlarmsAlarmCommandPayload  *psAlarmCommandPayload;
  tsCLD_AlarmsGetAlarmResponsePayload   *psGetAlarmResponse;
  } uMessage;
} tsCLD_AlarmsCallBackMessage;
```

When an Alarms event occurs, one of a number of command types could have been received. The relevant command type is specified through the u8CommandId field of the tsCLD_AlarmsCallBackMessage structure. The possible command types are detailed below.

The table below lists and describes the command types that can be received by the cluster server.

u8CommandId Enumeration	Description
E_CLD_ALARMS_CMD_RESET_ALARM	A Reset Alarm command has been received
E_CLD_ALARMS_CMD_RESET_ALL_ALARMS	A Reset All Alarms command has been received
E_CLD_ALARMS_CMD_GET_ALARM	A Get Alarm command has been received
E_CLD_ALARMS_CMD_RESET_ALARM_LOG	A Reset Alarm Log command has been received

The table below lists and describes the command types that can be received by the cluster client.

u8CommandId Enumeration	Description
E_CLD_ALARMS_CMD_ALARM	An Alarm notification has been received
E_CLD_ALARMS_CMD_GET_ALARM_RESPONSE	A Get Alarm response has been received

Parent topic:*Alarms Cluster*

Functions The following Alarms cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_AlarmsCreateAlarms
2. eCLD_AlarmsCommandResetAlarmCommandSend
3. eCLD_AlarmsCommandResetAllAlarmsCommandSend
4. eCLD_AlarmsCommandGetAlarmCommandSend
5. eCLD_AlarmsCommandResetAlarmLogCommandSend
6. eCLD_AlarmsResetAlarmLog
7. eCLD_AlarmsAddAlarmToLog
8. eCLD_AlarmsGetAlarmFromLog

eCLD_AlarmsCreateAlarms

```
teZCL_Status eCLD_AlarmsCreateAlarms(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits,
        tsCLD_AlarmsCustomDataStructure
    *psCustomDataStructure);
```

Description This function creates an instance of the Alarms cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function is only called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates an Alarms cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function is not called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be the first Alarms cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance*:Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Alarms cluster. This parameter can refer to a pre-filled structure called sCLD_``Alarms which is provided in the **Alarms.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_Alarms which defines the attributes of Alarms cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above)
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 17.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Functions

eCLD_AlarmsCommandResetAlarmCommandSend

```
teZCL_Status eCLD_AlarmsCommandResetAlarmCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_AlarmsResetAlarmCommandPayload
    *psPayload);
```

Description This function can be called on an Alarms cluster client to send a Reset Alarm command to a cluster server. This command requests that a specific alarm for a specific cluster is reset. The function may be called as the result of user input. The relevant alarm and cluster ID must be specified in the command payload (see Section 17.7.3.1).

On receiving the command, an E_CLD_ALARMS_CMD_RESET_ALARM event is generated on the cluster server to notify the application.

The function is only used to reset alarms that are not automatically reset when the alarm condition no longer exists.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for the command (see Section 17.7.3.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_AlarmsCommandResetAllAlarmsCommandSend

```
teZCL_Status eCLD_AlarmsCommandResetAllAlarmsCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be called on an Alarms cluster client to send a Reset All Alarms command to a cluster server. This command requests that all alarms on the server device are reset. The function may be called as the result of user input.

On receiving the command, an E_CLD_ALARMS_CMD_RESET_ALL_ALARMS event is generated on the cluster server to notify the application.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_AlarmsCommandGetAlarmCommandSend

```
teZCL_Status eCLD_AlarmsCommandGetAlarmCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on an Alarms cluster client to send a Get Alarm command to a cluster server. This command requests information on the logged alarm with the earliest timestamp in the device's Alarms table. As a result of this command, the retrieved entry is also deleted from the table.

The requested information is returned by the server in a Get Alarm response. When this response is received, an E_CLD_ALARMS_CMD_GET_ALARM_RESPONSE event is generated on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_AlarmsCommandResetAlarmLogCommandSend

```
teZCL_Status eCLD_AlarmsCommandResetAlarmLogCommandSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber);
```

Description This function can be called on an Alarms cluster client to send a Reset Alarm Log command to a cluster server. This command requests that the Alarms table on the server is cleared of all entries. The function may be called as the result of user input.

On receiving the command, an E_CLD_ALARMS_CMD_RESET_ALARM_LOG event is generated on the cluster server to notify the application but the ZCL automatically clears the Alarms table.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_AlarmsResetAlarmLog

```
teZCL_Status eCLD_AlarmsResetAlarmLog(
    tsZCL_EndPointDefinition *psEndPointDefinition,
    tsZCL_ClusterInstance *psClusterInstance);
```

Description This function can be called on the Alarms cluster server to clear all entries of the local Alarms table. The function may be called as the result of user input.

Parameters

- *psEndPointDefinition*: Pointer to the ZCL endpoint definition structure for the application (see Section 6.1.1)
- *psClusterInstance*: Pointer to structure containing information about the Alarms cluster instance (see Section 6.1.16)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_AlarmsAddAlarmToLog

```
teZCL_Status eCLD_AlarmsAddAlarmToLog(
    tsZCL_EndPointDefinition *psEndPointDefinition,
    tsZCL_ClusterInstance *psClusterInstance,
    uint8 u8AlarmCode,
    uint16 u16ClusterId);
```

Description This function can be called on the Alarms cluster server to add a new entry to the local Alarms table. The function should be called by the server application when an alarm condition has occurred. The alarm and the cluster which generated it must be specified. A timestamp (UTC) for the alarm is automatically inserted into the entry.

Parameters

- *psEndPointDefinition*: Pointer to the ZCL endpoint definition structure for the application (see Section 6.1.1)
- *psClusterInstance*: Pointer to structure containing information about the Alarms cluster instance (see Section 6.1.16)
- *u8AlarmCode*: Code that identifies the type of alarm to be added
- *u16ClusterId*: Cluster ID of the cluster which generated the alarm

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_AlarmsGetAlarmFromLog

```
teZCL_Status eCLD_AlarmsGetAlarmFromLog(
  tsZCL_EndPointDefinition *psEndPointDefinition,
  tsZCL_ClusterInstance *psClusterInstance,
  uint8 *pu8AlarmCode,
  uint16 *pu16ClusterId,
  uint32 *pu32TimeStamp);
```

Description This function can be called on the Alarms cluster server to obtain an entry from the local Alarms table. Information on the logged alarm with the earliest timestamp in the device's Alarms table is returned - pointers to memory locations to receive the retrieved alarm data must be provided. As a result of this command, the retrieved entry is also deleted from the table.

Parameters

- *psEndPointDefinition*: Pointer to the ZCL endpoint definition structure for the application (see Section 6.1.1)
- *psClusterInstance*: Pointer to structure containing information about the Alarms cluster instance (see Section 6.1.16)
- *pu8AlarmCode*: Pointer to location to receive the alarm code which identifies the retrieved alarm type
- *pu16ClusterId*: Pointer to location to receive the Cluster ID of the cluster which generated the alarm
- *pu32TimeStamp*: Pointer to location to receive timestamp (UTC) of the retrieved alarm (a value of 0XFFFFFFFF indicates that no timestamp is available for the alarm)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND

- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_AlarmsSignalAlarm

```
teZCL_Status eCLD_AlarmsSignalAlarm(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint8 u8AlarmCode,
    uint16 u16ClusterId);
```

Description This function can be called on the Alarms cluster server to send an Alarm notification to a cluster client and add a log entry to the local Alarms table on the server. The function should be called by the server application when an alarm condition has occurred. The alarm and the cluster which generated it must be specified.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *u8AlarmCode*: Code which identifies the type of alarm that has occurred
- *u16ClusterId*: Cluster ID of the cluster which generated the alarm

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

Parent topic:*Alarms Cluster*

Structures

Event Callback Message Structure For an Alarms event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_AlarmsCallBackMessage structure:

```
typedef struct
{
   uint8  u8CommandId;
   union
   {
 tsCLD_AlarmsResetAlarmCommandPayload *psResetAlarmCommandPayload;
 tsCLD_AlarmsAlarmCommandPayload    *psAlarmCommandPayload;
 tsCLD_AlarmsGetAlarmResponsePayload  *psGetAlarmResponse;
} uMessage;
} tsCLD_AlarmsCallBackMessage;
```

where:

- u8CommandId indicates the type of Alarms command that has been received by a cluster server or client, one of:
  - E_CLD_ALARMS_CMD_RESET_ALARM (server event)
  - E_CLD_ALARMS_CMD_RESET_ALL_ALARMS (server event)
  - E_CLD_ALARMS_CMD_GET_ALARM (server event)
  - E_CLD_ALARMS_CMD_RESET_ALARM_LOG (server event)
  - E_CLD_ALARMS_CMD_ALARM (client event)
  - E_CLD_ALARMS_CMD_GET_ALARM_RESPONSE (client event)
- uMessage is a union containing the command payload in the following form:
  - psResetAlarmCommandPayload is a pointer to a structure containing the Reset Alarm command payload - see Section 17.7.3.1
  - psAlarmCommandPayload is a pointer to a structure containing the Alarm notification payload - see Section 17.7.3.2
  - psGetAlarmResponse is a pointer to a structure containing the Get Alarm response payload - see Section 17.7.4.1

For further information on the above events, refer to Section 17.5.

Parent topic: Structures

Custom Data Structure The Alarms cluster requires extra storage space to be allocated for use by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
  DLIST    lAlarmsAllocList;
  DLIST    lAlarmsDeAllocList;
  tsZCL_ReceiveEventAddress        sReceiveEventAddress;
  tsZCL_CallBackEvent              sCustomCallBackEvent;
  tsCLD_AlarmsCallBackMessage      sCallBackMessage;
  tsCLD_AlarmsTableEntry
  asAlarmsTableEntry[CLD_ALARMS_MAX_NUMBER_OF_ALARMS];
} tsCLD_AlarmsCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Custom Command Payloads This section contains the structures for the payloads of the Alarms cluster custom commands.

Reset Alarm Command Payload

```
typedef struct
{
   uint8    u8AlarmCode;
   uint16   u16ClusterId;
} tsCLD_AlarmsResetAlarmCommandPayload;
```

where:

- u8AlarmCode is the code which identifies the type of alarm to be reset - these codes are cluster-specific
- u16ClusterId is the Cluster ID of the cluster which generates the alarm to be reset

Parent topic: Custom Command Payloads

Alarm Notification Payload

```
typedef struct
{
  uint8    u8AlarmCode;
  uint16    u16ClusterId;
} tsCLD_AlarmsAlarmCommandPayload;
```

where:

- u8AlarmCode is the code which identifies the type of alarm that has been generated - these codes are cluster-specific
- u16ClusterId is the Cluster ID of the cluster which generated the alarm

Parent topic: Custom Command Payloads

Parent topic: Structures

Custom Response Payloads This section contains the structures for the payloads of the Alarms cluster custom responses.

Get Alarm Response Payload

```
typedef struct
{
   uint8    u8Status;
   uint8    u8AlarmCode;
   uint16   u16ClusterId;
   uint32   u32TimeStamp;
} tsCLD_AlarmsGetAlarmResponsePayload;
```

where:

- u8Status indicates the result of the Get Alarm operation as follows:
  - SUCCESS (0x01): An alarm entry is successfully retrieved from the Alarms table and its details are reported in the remaining fields (below)
  - NOT_FOUND (0x00): There were no alarm entries to be retrieved from the Alarms table and the remaining fields (below) are empty
- u8AlarmCode is the code which identifies the type of alarm reported - these codes are cluster-specific
- u16ClusterId is the Cluster ID of the cluster which generated the alarm
- u32TimeStamp is a timestamp representing the time (UTC) at which the alarm was generated (a value of 0XFFFFFFFF indicates that no timestamp is available for the alarm)

Parent topic: Custom Response Payloads

Parent topic: Structures

Alarms Table Entry The following structure contains the data for an entry of an Alarms table.

```
typedef struct
{
    DNODE     dllAlarmsNode;
    uint8     u8AlarmCode;
    uint16    u16ClusterId;
    uint32    u32TimeStamp;
} tsCLD_AlarmsTableEntry;
```

where:

- dllAlarmsNode is for internal use and no knowledge of it is required
- u8AlarmCode is the code which identifies the type of alarm - these codes are cluster-specific
- u16ClusterId is the Cluster ID of the cluster which generated the alarm
- u32TimeStamp is a timestamp representing the time (UTC) at which the alarm was generated (a value of 0XFFFFFFFF indicates that no timestamp is available for the alarm)

Parent topic: Structures

Parent topic:*Alarms Cluster*

Enumerations

teCLD_Alarms_AttributeID The following structure contains the enumerations used to identify the attributes of the Alarms cluster.

```
typedef enum
{
  E_CLD_ALARMS_ATTR_ID_ALARM_COUNT = 0x0000,
} teCLD_Alarms_AttributeID;
```

Parent topic: Enumerations

Parent topic:*Alarms Cluster*

Time Cluster and ZCL Time This chapter describes the Time cluster which is defined in the ZCL. This cluster is used to maintain a time reference for the transactions in a ZigBee PRO network and to time synchronize the ZigBee PRO devices.

The Time cluster has a Cluster ID of 0x000A.

This section also describes the maintenance of 'ZCL time'.

Overview The Time cluster is required in a ZigBee PRO network in which the constituent devices must be kept time-synchronized - for example, in an HVAC system, it may be necessary for heating to operate only between specific times of the day. In such a case, one device implements the Time cluster as a server and acts as the time-master for the network. While the other devices in the network, implement the Time cluster as a client and time-synchronize with the server.

Note: As for all clusters, the Time cluster is stored in a shared device structure (see Section 18.3) which, for the cluster client, reflects the state of the cluster server. Access to the shared device structure (on Time cluster server and client) must be controlled using a mutex - for information on mutexes, refer to *Appendix A*.

The Time cluster is enabled by defining CLD_TIME in the **zcl_options.h** file. The inclusion of the client or server software must also be pre-defined in the application's compile-time options. In addition, if the cluster is to reside on a custom endpoint then specify the role of client or server, when creating the cluster instance. The compile-time options for the Time cluster are fully detailed in Section 18.10.

In addition to the time in the Time cluster, the ZCL also keeps its own time, 'ZCL time'. ZCL time may be maintained on a device even when the Time cluster is not used by the device. Both times are described below.

Time Attribute The Time cluster contains an attribute for the current time, as well as associated information such as time-zone and daylight saving - see Section 18.3. The time attribute is referenced to UTC (Coordinated Universal Time) and based on the type **UTCTime**, which is defined in the ZigBee standard as:

"UTCTime is an unsigned 32-bit value representing the number of seconds since 0 hours, 0 minutes, 0 seconds, on the January 1, 2000 UTC".

ZCL Time 'ZCL time' is based on the above **UTCTime** definition. This time is derived from a 1 second timer and is used to drive any ZCL timers that have been registered.

Parent topic:*Time Cluster and ZCL Time*

Time Cluster structure and attributes The Time cluster is contained in the following tsCLD_Time structure:

```
typedef struct
{
#ifdef TIME_SERVER
    zutctime                utctTime;
    zbmap8                  u8TimeStatus;
#ifdef CLD_TIME_ATTR_TIME_ZONE
    zint32                  i32TimeZone;
#endif
#ifdef CLD_TIME_ATTR_DST_START
    zuint32                 u32DstStart;
#endif
#ifdef CLD_TIME_ATTR_DST_END
    zuint32                 u32DstEnd;
#endif
#ifdef CLD_TIME_ATTR_DST_SHIFT
    zint32                  i32DstShift;
#endif
#ifdef CLD_TIME_ATTR_STANDARD_TIME
    zuint32                 u32StandardTime;
#endif
#ifdef CLD_TIME_ATTR_LOCAL_TIME
    zuint32                 u32LocalTime;
#endif
#ifdef CLD_TIME_ATTR_LAST_SET_TIME
    zutctime                u32LastSetTime;
#endif
#ifdef CLD_TIME_ATTR_VALID_UNTIL_TIME
    zutctime                u32ValidUntilTime;
#endif
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_Time;;
```

where:

- utctTime is a mandatory 32-bit attribute which holds the current time (UTC). This attribute can only be over-written using a remote 'write attributes' request if the local Time cluster is not configured as the time-master for the network - this is the case if bit 0 of the element u8TimeStatus (see below) is set to 0.
- u8TimeStatus is a mandatory 8-bit attribute containing the following bitmap:

Bits	Meaning	Description
0	Master	1: Time-master for network

0: Not time-master for network

| |1|Synchronized|1: Synchronized to another device

0: Not synchronized to another device

| |2|Master for Time Zone and DST *|1: Master for time-zone and DST

0: Not master for time-zone and DST

| |3-7|Reserved|-|

```
\* DST= Daylight Saving Time

Macros are provided for setting the individual bits of this bitmap:
```

(continues on next page)

(continued from previous page)

```
- CLD\_TM\_TIME\_STATUS\_MASTER\_MASK \(bit 0\)
- CLD\_TM\_TIME\_STATUS\_SYNCHRONIZED\_MASK \(bit 1\)
- CLD\_TM\_TIME\_STATUS\_MASTER\_ZONE\_DST\_MASK \(bit 2\)
```

- i32TimeZone is an optional attribute which indicates the local time-zone expressed as an offset from UTC, in seconds.
- u32DstStart is an optional attribute which contains the start-time (UTC), in seconds, for daylight saving for the current year.
- u32DstEnd is an optional attribute which contains the end-time (UTC), in seconds, for daylight saving for the current year.
- i32DstShift is an optional attribute which contains the local time-shift, in seconds, relative to standard local time that is applied during the daylight saving period.
- u32StandardTime is an optional attribute which contains the local standard time (equal to utctTime + i32TimeZone).
- u32LocalTime is an optional attribute which contains the local time taking into account daylight saving, if applicable (equal to utctTime + i32TimeZone + i32DstShift during the daylight saving period).
- u32LastSetTime is an optional attribute which indicates the most recent UTC time at which the Time attribute (utctTime) was set, either internally or over the ZigBee network.
- u32ValidUntilTime is an optional attribute which indicates a UTC time (later than u32LastSetTime) up to which the Time attribute (utctTime) value may be trusted.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Note: If necessary, the daylight saving attributes (u32DstStart, u32DstEnd and, i32DstShift) must all be enabled together.

The Time cluster structure contains two mandatory elements, utctTime and u8TimeStatus. The remaining elements are optional, each being enabled/disabled through a corresponding macro defined in the **zcl_options.h** file - for example, the optional time zone element i32TimeZone is enabled/disabled through the macro CLD_TIME_ATTR_TIME_ZONE (see Section 18.3.2).

Parent topic:*Time Cluster and ZCL Time*

Attribute Settings

Mandatory Attributes The mandatory attributes of the Time cluster are set as follows:

utctTime This is a mandatory 32-bit attribute which holds the current time (UTC). On the time-master, this attribute value is incremented once per second. On all other devices, it is the responsibility of the local application to synchronise this time with the time-master. For more information on time-synchronisation, refer to Section 18.5.

u8TimeStatus This is a mandatory 8-bit attribute containing the bitmap detailed in Table 28 onpage 389. This attribute must be set as follows on the time-master (Time cluster server):

- The 'Master' bit should initially be zero until the current time has been obtained from an external time-of-day source. Once the time has been obtained and set, the 'Master' bit should be set (to '1').

- The 'Synchronised' bit must always be zero, as the time-master does not obtain its time from another device within the ZigBee network (this bit is set to '1' only for a secondary Time cluster server that is synchronized to the time-master).
- The 'Master for Time Zone and DST' bit must be set (to '1') once the time-zone and Daylight Saving Time (DST) attributes (see below) have been correctly set for the device.

Macros are provided for setting the individual bits of the u8TimeStatus bitmap - for example, the macro CLD_TM_TIME_STATUS_MASTER_MASK is used to set the Master bit. These macros are defined in the header file **time.h** and are also listed in Section 18.2.

Parent topic: Attribute Settings

Optional Attributes The optional attributes of the Time cluster are set as follows:

i32TimeZone This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_TIME_ZONE and which indicates the local time-zone.

The local time-zone is expressed as an offset from UTC, where this offset is quantified in seconds. Therefore:

Current local standard time = utctTime + i32TimeZone

where i32TimeZone is negative if the local time is behind UTC.

u32DstStart This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_DST_START and which contains the start-time (in seconds) for daylight saving for the current year.

If u32DstStart is used then u32DstEnd and i32DstShift are also required.

u32DstEnd This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_DST_END and which contains the end-time (in seconds) for daylight saving for the current year.

If u32DstEnd is used then u32DstStart and i32DstShift are also required.

i32DstShift This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_DST_SHIFT and which contains the local time-shift (in seconds), relative to standard local time, that is applied during the daylight saving period (between u32DstStart and u32DstEnd). During this period:

Current local time = utctTime + i32TimeZone + i32DstShift

This time-shift varies between territories, but is 3600 seconds (1 hour) for Europe and North America.

If i32DstShift is used then u32DstStart and u32DstEnd are also required.

u32StandardTime This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_STANDARD_TIME and which contains the local standard time (equal to utctTime + i32TimeZone).

u32LocalTime This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_LOCAL_TIME and which contains the local time taking into account daylight saving, if applicable (equal to utctTime + i32TimeZone + i32DstShift during the daylight saving period and equal to u32StandardTime outside of the daylight saving period).

u32LastSetTime This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_LAST_SET_TIME and which indicates the most recent UTC time at which the Time attribute (utctTime) was set, either internally or over the ZigBee network.

u32ValidUntilTime This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_VALID_UNTIL_TIME and indicates a UTC time (later than u32LastSetTime) up to which the Time attribute (utctTime) value may be trusted.

Parent topic: Attribute Settings

Parent topic:*Time Cluster and ZCL Time*

Maintaining ZCL Time The simplest case of keeping time on a ZigBee PRO device is to maintain 'ZCL time' only (without using the Time cluster). In this case, the ZCL time on a device can be initialized by the application using the function **vZCL_SetUTCTime()**.

The ZCL time is subsequently incremented from a local one-second timer, as follows. On expiration of the timer, an event is generated (from the hardware/software timer that drives the one-second timer), which causes a ZCL user task to be activated. The event is initially handled by this task as described in Section 3.2, resulting in an E_ZCL_CBET_TIMER event being passed to the ZCL via the function **vZCL_EventHandler()**. The following actions should then be performed:

1. The ZCL automatically increments the ZCL time and may run cluster-specific schedulers.
2. The user task resumes the one-second timer.

Updating ZCL Time Following Sleep In the case of a device that sleeps, on waking from sleep, the application should update the ZCL time using the function **vZCL_SetUTCTime()**according to the duration for which the device was asleep. This requires the sleep duration to be timed.

While sleeping, the device normally uses its RC oscillator for timing purposes, which may not maintain the required accuracy. It is therefore recommended that a more accurate external crystal is used to time the sleep periods.

The **vZCL_SetUTCTime()** function does not cause timer events to be executed. If the device is awake for less than one second, the application should generate a E_ZCL_CBET_TIMER event to prompt the ZCL to run any timer-related functions. Note that when passed into **vZCL_EventHandler()**, this event will increment the ZCL time by one second.

Parent topic: Maintaining ZCL Time

ZCL Time Synchronization The local ZCL time on a device can be synchronized with the time in a time-related cluster, such as Time, Price, or Messaging. The ZCL time is considered to be synchronized following a call to **vZCL_SetUTCTime()**. The NXP implementation of the ZCL also provides the following functions relating to ZCL time synchronization:

- **u32ZCL_GetUTCTime()** obtains the ZCL time (held locally).
- **bZCL_GetTimeHasBeenSynchronised()** determines whether the ZCL time on the device has been synchronized - that is, whether **vZCL_SetUTCTime()** has been called.
- **vZCL_ClearTimeHasBeenSynchronised()** can be used to specify that the device can no longer be considered to be synchronized (for example, if there has been a problem in accessing the Time cluster server over a long period).

Parent topic: Maintaining ZCL Time

Parent topic:*Time Cluster and ZCL Time*

Time-Synchronization of Devices The devices in a ZigBee PRO network may need to be time-synchronized (so that they all refer to the same time). In this case, the Time cluster is used and one device acts as the Time cluster server and time-master from which the other devices set their time.

Note: Synchronization with a time-master is not required in all networks. In such cases, it is sufficient to use the ZCL time without synchronization between devices, as described in Section 18.4.

There are two times on a device that should be maintained during the synchronisation process:

- Time attribute of the Time cluster (utctTime field of tsCLD_Time structure)
- ZCL time

On the time-master, these times are initialized by the local application using an external master time and are subsequently maintained using a local one-second timer (see Section 18.5.1), as well as occasional re-synchronizations with external master time.

On all other devices, these times are initialized by the local application by synchronizing with the time-master (see Section 18.5.2). The ZCL time is subsequently maintained using a local one-second timer and both times are occasionally re-synchronized with the time-master (see Section 18.5.3).

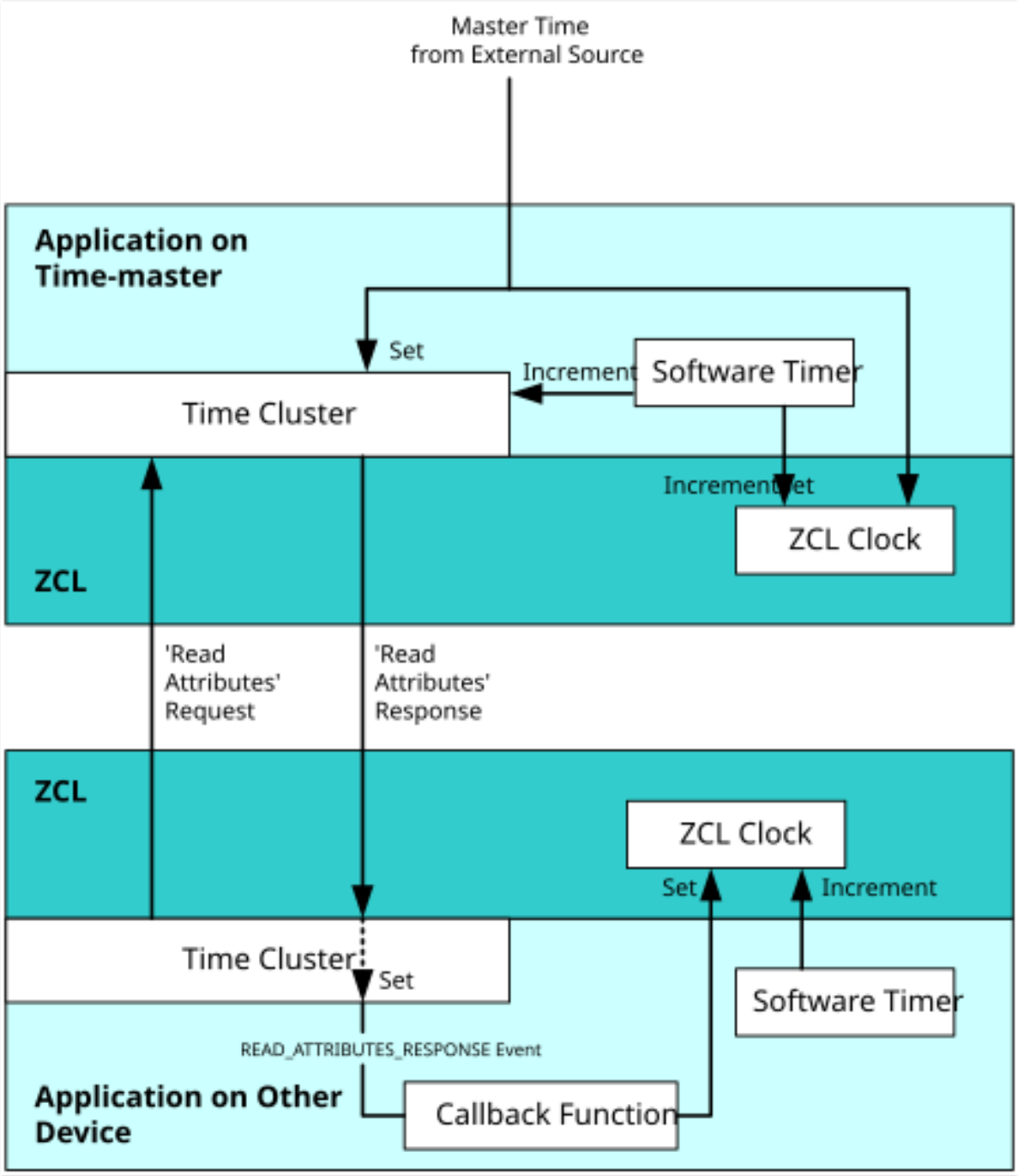
synchronization with the time-master is normally performed via the Time cluster.

CAUTION:

If there is more than one Time cluster server in the network, devices should only attempt to synchronize to one server in order to prevent their clocks from repeatedly jittering backwards and forwards.

The figure below provides an overview of the time initialization and synchronization processes described in the sub-sections that follow.

Time Initialization and Synchronization



Initialising and Maintaining Master Time The time-master must initially obtain a master time from an external source. The application on the time-master must use this time to set its ZCL time by calling the function **vZCL_SetUTCTime()** and to set the value of the Time cluster attribute utctTime in the local tsCLD_Time structure within the shared device structure (securing access with a mutex). The application must also set (to '1') the 'Master' bit of the u8TimeStatus attribute of the tsCLD_Time structure, to indicate that this device is the time-master and that the time has been set.

Note: The 'Synchronised' bit of the u8TimeStatus attribute should always be zero on the time-master, as this device does not synchronise to any other device within the ZigBee network.

If the time-master has also obtained time-zone and daylight saving information (or has been pre-programmed with this information), its application must set (to '1') the 'Master for Time

Zone and DST' bit of the u8TimeStatus attribute and write the relevant optional attributes. These optional attributes can then be used to provide time-zone and daylight saving information to other devices (see Section 18.3).

Note: The time-master can prevent other devices from attempting to read its Time cluster attributes before the time has been set - the initialization of the master time should be done after registering the endpoint for the device and before starting the ZigBee PRO stack.

The ZCL time and the utctTime attribute are subsequently incremented from a local one-second timer, as follows. On expiration of the timer, an event is generated (from the hardware/software timer that drives the one-second timer), which causes a ZCL user task to be activated. The event is initially handled by this task as described in Section 3.2, resulting in an E_ZCL_CBET_TIMER event being passed to the ZCL via the function **vZCL_EventHandler()**. The following actions should then be performed:

1. The ZCL automatically increments the ZCL time and may run cluster-specific schedulers (e.g. for maintaining a price list).
2. The user task updates the value of the utctTime attribute of the tsCLD_Time structure within the shared device structure (securing access with a mutex).
3. The user task resumes the one-second timer.

Both the ZCL time and the utctTime attribute must also be updated by the application when an update of the master time is received.

Parent topic: Time-Synchronization of Devices

Initial Synchronisation of Devices It is the responsibility of the application on a ZigBee PRO device to perform time-synchronisation with the time-master. The application can remotely read the Time cluster attributes from the time-master by calling the function **eZCL_SendReadAttributesRequest()**, which will result in a 'read attributes' response containing the Time cluster data. On receiving this response, a 'data indication' stack event is generated on the local device, which causes a ZCL user task to be activated. The event is initially handled by this task as described in Section 3.2, resulting in an E_ZCL_ZIGBEE_EVENT event being passed to the ZCL via the function **vZCL_EventHandler()**. Provided that the event contains a message incorporating a 'read attributes' response, the ZCL:

1. automatically sets the utctTime field of the tsCLD_Time structure to the value of the same attribute in the 'read attributes' response (and also sets other Time cluster attributes, if requested)
2. invokes the relevant user-defined callback function (see *Chapter 3*), which must read the local utctTime attribute (securing access with a mutex) and use this value to set the ZCL time by calling the function **vZCL_SetUTCTime()**

Note: When a device attempts to time-synchronise with the time-master, it should check the u8TimeStatus attribute in the 'read attributes' response. If the 'Master' bit of this attribute is not equal to '1', the obtained time should not be trusted and the time should not be set. The device should wait and try to synchronise again later.

It may also be possible to obtain time-zone and daylight saving information from the time-master. If available, this information will be returned in the 'read attributes' response. However, before using these optional Time cluster attributes from the response, the application should first check that the 'Master for Time Zone and DST' bit of the u8TimeStatus attribute is set (to '1') in the response.

The ZCL time and utctTime attribute value on the local device are subsequently maintained as described in Section 18.5.3.

Parent topic: Time-Synchronization of Devices

Re-synchronisation of Devices After the initialization described in Section 18.5.2, the ZCL time must be updated by the application on each one-second tick of the local timer. The ZCL time is updated from the timer in the same way as described in Section 18.4.

Due to the inaccuracy of the local one-second timer, the ZCL time is likely to lose synchronisation with the time on the time-master. It will therefore be necessary to occasionally re-synchronise the local ZCL time with the time-master - the utctTime attribute value is also updated at the same time. A device can re-synchronise with the time-master by first remotely reading the utctTime attribute using the function **eZCL_SendReadAttributesRequest()**. On receiving the 'read attributes' response from the time-master, the operations performed are the same as those described for initial synchronisation in Section 18.5.2.

Parent topic: Time-Synchronization of Devices

Parent topic:*Time Cluster and ZCL Time*

Time Event The Time cluster does not have any events of its own, but the ZCL includes one time-related event: E_ZCL_CBET_TIMER. For this event, the eEventType field of the tsZCL_CallBackEvent structure (see Section 3.1) is set to E_ZCL_CBET_TIMER.

The application may need to generate this event, as indicated in Section 3.2.

Parent topic:*Time Cluster and ZCL Time*

Functions The following time-related functions are provided in the NXP implementation of the ZCL:

1. eCLD_TimeCreateTime
2. vZCL_SetUTCTime
3. u32ZCL_GetUTCTime
4. bZCL_GetTimeHasBeenSynchronised
5. vZCL_ClearTimeHasBeenSynchronised

Note: The time used in the Time cluster and in the ZCL is a UTC (Co-ordinated Universal Time) type **UTCTime**, which is defined in the ZigBee Specification as follows: *"UTCTime is an unsigned 32 bit value representing the number of seconds since 0 hours, 0 minutes, 0 seconds, on the 1st of January, 2000 UTC"*

eCLD_TimeCreateTime

```
teZCL_Status eCLD_TimeCreateTime(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Time cluster on the local endpoint. The cluster instance can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Time cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device (e.g. Simple Sensor) will be used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be the first Time cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Time cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Time cluster. This parameter can refer to a pre-filled structure called sCLD_Time which is provided in the **Time.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_Time which defines the attributes of Time cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

vZCL_SetUTCTime

```
void vZCL_SetUTCTime(uint32 u32UTCTime);
```

Description This function sets the current time (UTC) that is stored in the ZCL ('ZCL time').

The application may call this function, for example, when a time update has been received (e.g. via the Time or Price cluster).

Note that this function does not update the time in the Timer cluster - if required, the application must do this by writing to the tsCLD_Time structure (see Section 18.2).

Parameters

- *u32UTCTime* The current time (UTC) to be set, in seconds

Returns

- None

Parent topic: Functions

u32ZCL_GetUTCTime

```
uint32 u32ZCL_GetUTCTime(void);
```

Description This function obtains the current time (UTC) that is stored in the ZCL ('ZCL time').

Parameters

- None

Returns

- The current time (UTC), in seconds, obtained from the ZCL

Parent topic: Functions

bZCL_GetTimeHasBeenSynchronised

```
bool_t bZCL_GetTimeHasBeenSynchronised(void);
```

Description This function queries whether the ZCL time on the device has been synchronized.

The clock is considered to be unsynchronized at start-up and is synchronized following a call to **vZCL_SetUTCtime()**. The ZCL time must be synchronized before using the time-related functions of other clusters.

Parameters

- None

Returns

- TRUE if the local ZCL time has been synchronized, otherwise FALSE

Parent topic: Functions

vZCL_ClearTimeHasBeenSynchronised

```
void vZCL_ClearTimeHasBeenSynchronised(void);
```

Description This function is used to notify the ZCL that the local ZCL time may no longer be accurate.

Parameters

- None

Returns

- None

Parent topic: Functions

Parent topic:*Time Cluster and ZCL Time*

Return codes The time-related functions use the ZCL return codes defined in Section 7.2.

Parent topic:*Time Cluster and ZCL Time*

Enumerations

teCLD_TM_AttributeID The following structure contains the enumerations used to identify the attributes of the Time cluster.

```
typedef enum
{
 E_CLD_TIME_ATTR_ID_TIME  = 0x0000, /* Mandatory */
 E_CLD_TIME_ATTR_ID_TIME_STATUS, /* Mandatory */
 E_CLD_TIME_ATTR_ID_TIME_ZONE,
 E_CLD_TIME_ATTR_ID_DST_START,
 E_CLD_TIME_ATTR_ID_DST_END,
 E_CLD_TIME_ATTR_ID_DST_SHIFT,
 E_CLD_TIME_ATTR_ID_STANDARD_TIME,
 E_CLD_TIME_ATTR_ID_LOCAL_TIME,
 E_CLD_TIME_ATTR_ID_LAST_SET_TIME,
 E_CLD_TIME_ATTR_ID_VALID_UNTIL_TIME
} teCLD_TM_AttributeID;
```

Parent topic: Enumerations

Parent topic:*Time Cluster and ZCL Time*

Compile-time Options To enable the Time cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_TIME
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define TIME_CLIENT
#define TIME_SERVER
```

The Time cluster contains macros that may be optionally specified at compile-time by adding some or all of the following lines to the **zcl_options.h** file.

Add this line to enable the optional Time Zone attribute

```
#define CLD_TIME_ATTR_TIME_ZONE
```

Add this line to enable the optional DST Start attribute

```
#define CLD_TIME_ATTR_DST_START
```

Add this line to enable the optional DST End attribute

```
#define CLD_TIME_ATTR_DST_END
```

Add this line to enable the optional DST Shift attribute

```
#define CLD_TIME_ATTR_DST_SHIFT
```

Add this line to enable the optional Standard Time attribute

```
#define CLD_TIME_ATTR_STANDARD_TIME
```

Add this line to enable the optional Local Time attribute

```
#define CLD_TIME_ATTR_LOCAL_TIME
```

Note: Some attributes must always be enabled together - for example, if daylight saving is to be implemented then CLD_TIME_ATTR_DST_START, CLD_TIME_ATTR_DST_END and CLD_TIME_ATTR_DST_SHIFT must all be included in the **zcl_options.h** file.

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_TIME_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Time Cluster and ZCL Time*

Input and Output Clusters This chapter details the following input and output clusters:

- Analogue Input (Basic) - see Section 19.1
- Analogue Output (Basic) - see Section 19.2
- Binary Input (Basic) - see Section 19.3
- Binary Output (Basic) - see Section 19.4
- Multistate Input (Basic) - see Section 19.5
- Multistate Output (Basic) - see Section 19.6

Analogue Input (Basic) This chapter describes the Analogue Input (Basic) cluster, which provides an interface for accessing an analogue measurement.

The Analogue Input (Basic) cluster has a Cluster ID of 0x000C.

Overview The Analogue Input (Basic) cluster provides an interface for accessing an analogue measurement and its associated characteristics. It is typically used in a sensor that measures an analogue physical quantity.

To use the functionality of this cluster, you must include the file **AnalogInputBasic.h** in your application and enable the cluster by defining CLD_ANALOG_INPUT_BASIC in the **zcl_options.h** file.

An Analogue Input (Basic) cluster instance can act as either a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Analogue Input (Basic) cluster are fully detailed in Section 19.1.6.

Parent topic: Analogue Input (Basic)

Analogue Input (Basic) Structure and Attributes The structure definition for the Analogue Input (Basic) cluster is:

```
typedef struct
{
#ifdef ANALOG_INPUT_BASIC_SERVER
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_DESCRIPTION
    tsZCL_CharacterString        sDescription;
    zuint8     au8Description[16];
#endif
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_MAX_PRESENT_VALUE
    zsingle    fMaxPresentValue;
#endif
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_MIN_PRESENT_VALUE
    zsingle    fMinPresentValue;
#endif
    zbool      bOutOfService;
    zsingle    fPresentValue;
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_RELIABILITY
    zenum8     u8Reliability;
#endif
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_RESOLUTION
    zsingle    fResolution;
#endif
    zbmap8     u8StatusFlags;
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_ENGINEERING_UNITS
    zenum16    u16EngineeringUnits;
#endif
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_APPLICATION_TYPE
    zuint32    u32ApplicationType;
#endif
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8     u8AttributeReportingStatus;
#endif
#endif
    zuint16    u16ClusterRevision;
} tsCLD_AnalogInputBasic;
```

- The following optional pair of attributes are used to store a human readable description of the usage of the analogue input (for example, "Kitchen Temp"):
  - sDescription is a tsZCL_CharacterString structure (see Section6.1.14) for a string of up to 16 characters representing the description

  - au8Description[16] is a byte-array which contains the character data bytes representing the description
  - fMaxPresentValue is an optional attribute which indicates the highest analogue input value that can be reliably obtained and stored in the fPresentValue attribute.
- fMinPresentValue is an optional attribute which indicates the lowest analogue input value that can be reliably obtained and stored in the fPresentValue attribute.
- bOutOfService is a mandatory attribute which indicates whether the analogue input is currently in or out of service:
  - TRUE: Out of service
  - FALSE In service
    - If this attribute is set to TRUE, the fPresentValue attribute will not be updated to contain the current value of the input.
- fPresentValue is a mandatory attribute representing the latest analogue input value (this attribute is updated when the analogue input is re-sampled).
- u8Reliability is an optional attribute which indicates whether the value reported through fPresentValue is reliable or why it might be unreliable:
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_SENSOR
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_OVER_RANGE
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_UNDER_RANGE
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_OPEN_LOOP
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_SHORTED_LOOP
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_OUTPUT
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_PROCESS_ERROR
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
- fResolution is an optional attribute which indicates the smallest detectable change in the analogue input value that will result in an update of the attribute fPresentValue.
- u8StatusFlags is a mandatory attribute which is a bitmap representing the following status flags:

Bits	Name	Description
0	In Alarm	Reserved - unused for Analogue Input (Basic) cluster
1	Fault	- 1: Optional attribute u8Reliability is used and does not have a value of NO_FAULT_DETECTED

- 0: Otherwise

| |2|Overridden|- 1: Cluster has been overridden by a local mechanism

(fP``resentValue and u8Reliability will not track input)

- 0: Otherwise

| |3|Out Of Service|- 1: Attribute bOutOfService is set to TRUE

- 0: Otherwise

| |4-7|-|Reserved|

- u16EngineeringUnits is an optional attribute which indicates the physical unit of measure for the analogue input value recorded in the attribute fPresentValue. The values 0x0000 to 0x00FE are used to represent the units specified in Clause 21 of the BACnet standard. The value 0x00FF represents 'other' unit, and the values 0x0100 to 0xFFFF are for proprietary use. If the attribute u32ApplicationType is used and specifies an application type with an associated unit of measure, this unit will take precedence over the one specified in u16EngineeringUnits.
- u32ApplicationType is an optional attribute which is a bitmap representing the application type, as follows:

Bits	Field Name	Description
0-15	Index	Specific application usage (for example,Boiler Entering Temperature). There is a set of possible usages for each value of Type (see below). For these lists, refer to the attribute description in the ZCL Specification.
16-23	Type	Physical quantity measured (for example,Temperature). For the Analogue Input cluster, this can be a value in the range 0x00 to 0x0E. For the corresponding quantities, refer to the attribute description in the ZCL Specification.
24-31	Group	Identifier for the cluster that this attribute is part of (not the Cluster ID). For the Analogue Input cluster, this is 0x00.

- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic: Analogue Input (Basic)

Attributes for Default Reporting The following attributes of the Analogue Input (Basic) cluster can be selected for default reporting:

```
fPresentValue
```

- u8AttributeReportingStatus

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic: Analogue Input (Basic)

Functions The following Analogue Input (Basic) cluster function is provided in the NXP implementation of the ZCL:

eCLD_AnalogInputBasicCreateAnalogInputBasic

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

eCLD_AnalogInputBasicCreateAnalogInputBasic

```
teZCL_Status eCLD_AnalogInputBasicCreateAnalogInputBasic(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Analogue Input (Basic) cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create an Analogue Input (Basic) cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Analog Input Basic cluster.

The function initializesthe array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Analogue Input (Basic) cluster. This parameter can refer to a pre-filled structure called sCLD_AnalogInputBasic which is provided in the **AnalogInputBasic.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_AnalogInputBasic which defines the attributes of Analogue Input (Basic) cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic: Analogue Input (Basic)

Enumerations

teCLD_AnalogInputBasicCluster_AttrID The following structure contains the enumerations used to identify the attributes of the Analogue Input (Basic) cluster.

```
typedef enum
{
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_DESCRIPTION,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_MAX_PRESENT_VALUE,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_MIN_PRESENT_VALUE,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_OUT_OF_SERVICE,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_PRESENT_VALUE,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_RELIABILITY,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_RESOLUTION,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_STATUS_FLAGS,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_ENGINEERING_UNITS,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_APPLICATION_TYPE,
} teCLD_AnalogInputBasicCluster_AttrID;
```

Parent topic: Enumerations

teCLD_AnalogInputBasic_Reliability The following structure contains the enumerations used to report the value of the u8Reliability attribute (see Section 19.1.2).

```
typedef enum
{
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_SENSOR,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_OVER_RANGE,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_UNDER_RANGE,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_OPEN_LOOP,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_SHORTED_LOOP,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_OUTPUT,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_PROCESS_ERROR,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
}teCLD_AnalogInputBasic_Reliability;
```

Parent topic: Enumerations

Parent topic: Analogue Input (Basic)

Compile-time Options To enable the Analogue Input (Basic) cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_ANALOG_INPUT_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define ANALOG_INPUT_BASIC_CLIENT
#define ANALOG_INPUT_BASIC_SERVER
```

Optional Attributes The optional attributes for the Analogue Input (Basic) cluster (see Section 19.1.2) are enabled by defining:

- CLD_ANALOG_INPUT_BASIC_ATTR_DESCRIPTION
- CLD_ANALOG_INPUT_BASIC_ATTR_MAX_PRESENT_VALUE
- CLD_ANALOG_INPUT_BASIC_ATTR_MIN_PRESENT_VALUE
- CLD_ANALOG_INPUT_BASIC_ATTR_RELIABILITY
- CLD_ANALOG_INPUT_BASIC_ATTR_RESOLUTION
- CLD_ANALOG_INPUT_BASIC_ATTR_ENGINEERING_UNITS
- CLD_ANALOG_INPUT_BASIC_ATTR_APPLICATION_TYPE
- CLD_ANALOG_INPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_ANALOG_INPUT_BASIC_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_ANALOG_INPUT_BASIC_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic: Analogue Input (Basic)

Parent topic:*Input and Output Clusters*

Analogue Output (Basic) This chapter describes the Analogue Output (Basic) cluster, which provides an interface for setting the value of an analogue output.

The Analogue Output (Basic) cluster has a Cluster ID of 0x000D.

Overview The Analogue Input (Basic) cluster provides an interface for setting the value of an analogue output and accessing its associated characteristics. It is typically used in a a controller that outputs an analogue control signal.

To use the functionality of this cluster, you must include the file **AnalogOutputBasic.h** in your application and enable the cluster by defining CLD_ANALOG_OUTPUT_BASIC in the **zcl_options.h** file.

An Analogue Output (Basic) cluster instance can act as either a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Analogue Output (Basic) cluster are fully detailed in Section 19.2.6.

Parent topic: Analogue Output (Basic)

Analogue Output (Basic) Structure and Attributes The structure definition for the Analogue Output (Basic) cluster is:

```
typedef struct
{
#ifdef ANALOG_OUTPUT_BASIC_SERVER
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_DESCRIPTION
  tsZCL_CharacterString  sDescription;
  uint8  au8Description[16];
#endif
#ifdef  CLD_ANALOG_OUTPUT_BASIC_ATTR_MAX_PRESENT_VALUE
  zsingle       fMaxPresentValue;
#endif
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_MIN_PRESENT_VALUE
  zsingle       fMinPresentValue;
#endif
  zbool         bOutOfService;
  zsingle       fPresentValue;
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_RELIABILITY
  zenum8        u8Reliability;
#endif
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_RELINQUISH_DEFAULT
  zsingle       fRelinquishDefault;
#endif
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_RESOLUTION
  zsingle       fResolution;
#endif
  zbmap8        u8StatusFlags;
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_ENGINEERING_UNITS
  zenum16       u16EngineeringUnits;
#endif
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_APPLICATION_TYPE
  zuint32       u32ApplicationType;
#endif
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
  zenum8        u8AttributeReportingStatus;
#endif
#endif
  zuint16       u16ClusterRevision;
} tsCLD_AnalogOutputBasic;
```

- The following optional pair of attributes store a human readable description of the usage of the analogue output (for example, “Fan Speed”):
  - sDescription is a tsZCL_CharacterString structure (see Section6.1.14) for a string of up to 16 characters representing the description:
  - au8Description[16] is a byte-array which contains the character data bytes representing the description
  - fMaxPresentValue is an optional attribute which indicates the highest analogue output value that can be reliably used and stored in the fPresentValue attribute.
  - fMinPresentValue is an optional attribute which indicates the lowest analogue output value that can be reliably used and stored in the fPresentValue attribute.
- bOutOfService is a mandatory attribute which indicates whether the analogue output is currently in or out of service:
  - TRUE: Out of service
  - FALSE: In service
    - If this attribute is set to TRUE, the fPresentValue attribute will not be used to control the physical analogue output.

- fPresentValue is a mandatory attribute representing the latest analogue output value (this attribute is used to control the physical analogue output).
- u8Reliability is an optional attribute which indicates whether the value contained in fPresentValue is reliable or why it might be unreliable:
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_OVER_RANGE
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_UNDER_RANGE
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_OPEN_LOOP
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_SHORTED_LOOP
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_PROCESS_ERROR
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
- fRelinquishDefault is an optional attribute representing the default value to be used for fPresentValue when the supplied value is invalid.
- fResolution is an optional attribute which indicates the smallest change in the analogue output value that will cause the application to update the attribute fPresentValue.
- u8StatusFlags is a mandatory attribute which is a bitmap representing the following status flags:

Bits	Name	Description
0	In Alarm	Reserved - unused for Analogue Output (Basic) cluster
1	Fault	- 1: Optional attribute u8Reliability is used and does not have a value of NO_FAULT_DETECTED

- 0: Otherwise

| |2|Overridden|- 1: Cluster has been over-ridden by a local mechanism

(fP``resentValue and u8Reliability will not track input)

- 0: Otherwise

| |3|Out Of Service|- 1: Attribute bOutOfService is set to TRUE

- 0: Otherwise

| |4-7|-|Reserved|

- u16EngineeringUnits is an optional attribute which indicates the physical unit of measure for the analogue output value recorded in the attribute fPresentValue. The values 0x0000 to 0x00FE are used to represent the units specified in Clause 21 of the BACnet standard. The value 0x00FF represents 'other' unit, and the values 0x0100 to 0xFFFF are for proprietary use. If the attribute u32ApplicationType is used and specifies an application type with an associated unit of measure, this unit will take precedence over the one specified in u16EngineeringUnits.
- u32ApplicationType is an optional attribute which is a bitmap representing the application type, as follows:

Bits	Field Nam	Description
0-15	Index	Specific application usage (for example, Fan Speed). There is a set of possible usages for each value of Type (see below). For these lists, refer to the attribute description in the ZCL Specification.
16-23	Type	Physical quantity controlled (for example, Rotational Speed). For the Analogue Output cluster, this can be a value in the range 0x00 to 0x0E. For the corresponding quantities, refer to the attribute description in the ZCL Specification.
24-31	Grou	Identifier for the cluster that this attribute is part of (not the Cluster ID). For the Analogue Output cluster, this is 0x01.

- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic: Analogue Output (Basic)

Attributes for Default Reporting The following attributes of the Analogue Output (Basic) cluster can be selected for default reporting:

```
fPresentValue
u8AttributeReportingStatus
```

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic: Analogue Output (Basic)

Functions The following Analogue Output (Basic) cluster function is provided in the NXP implementation of the ZCL:

- eCLD_AnalogOutputBasicCreateAnalogOutputBasic

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

eCLD_AnalogOutputBasicCreateAnalogOutputBasic

```
teZCL_Status eCLD_AnalogOutputBasicCreateAnalogOutputBasic(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Analogue Output (Basic) cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create an Analogue Output (Basic) cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Analog Output Basic cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Analogue Output (Basic) cluster. This parameter can refer to a pre-filled structure called sCLD_AnalogOutputBasic which is provided in the **AnalogOutputBasic.h** file.
- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_AnalogOutputBasic which defines the attributes of Analogue Output (Basic) cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic: Analogue Output (Basic)

Enumerations

teCLD_AnalogOutputBasicCluster_AttrID The following structure contains the enumerations used to identify the attributes of the Analogue Output (Basic) cluster.

```
typedef enum
{
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_DESCRIPTION,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_MAX_PRESENT_VALUE,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_MIN_PRESENT_VALUE,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_OUT_OF_SERVICE,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_PRESENT_VALUE,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_RELIABILITY,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_RELINQUISH_DEFAULT,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_RESOLUTION,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_STATUS_FLAGS,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_ENGINEERING_UNITS,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_APPLICATION_TYPE,
} teCLD_AnalogOutputBasicCluster_AttrID;
```

Parent topic: Enumerations

teCLD_AnalogOutputBasic_Reliability The following structure contains the enumerations used to report the value of the u8Reliability attribute (see Section 19.2.2).

```
typedef enum
{
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_OVER_RANGE,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_UNDER_RANGE,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_OPEN_LOOP,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_SHORTED_LOOP,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_PROCESS_ERROR,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
}teCLD_AnalogOutputBasic_Reliability;
```

Parent topic: Enumerations

Parent topic: Analogue Output (Basic)

Compile-time options To enable the Analogue Output (Basic) cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_ANALOG_OUTPUT_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define ANALOG_OUTPUT_BASIC_CLIENT
#define ANALOG_OUTPUT_BASIC_SERVER
```

Optional Attributes The optional attributes for the Analogue Output (Basic) cluster (see Section 19.2.2) are enabled by defining:

- CLD_ANALOG_OUTPUT_BASIC_ATTR_DESCRIPTION
- CLD_ANALOG_OUTPUT_BASIC_ATTR_MAX_PRESENT_VALUE

- CLD_ANALOG_OUTPUT_BASIC_ATTR_MIN_PRESENT_VALUE
- CLD_ANALOG_OUTPUT_BASIC_ATTR_RELIABILITY
- CLD_ANALOG_OUTPUT_BASIC_ATTR_RELINQUISH_DEFAULT
- CLD_ANALOG_OUTPUT_BASIC_ATTR_RESOLUTION
- CLD_ANALOG_OUTPUT_BASIC_ATTR_ENGINEERING_UNITS
- CLD_ANALOG_OUTPUT_BASIC_ATTR_APPLICATION_TYPE
- CLD_ANALOG_OUTPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_ANALOG_OUTPUT_BASIC_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic: Analogue Output (Basic)

Parent topic:*Input and Output Clusters*

Binary Input (Basic) Cluster This chapter describes the Binary Input (Basic) cluster, which provides an interface for accessing a binary (two-state) measurement.

The Binary Input (Basic) cluster has a Cluster ID of 0x000F.

Overview The Binary Input (Basic) cluster provides an interface for accessing a binary measurement and its associated characteristics. It is typically used to implement a sensor that measures a two-state physical quantity.

To use the functionality of this cluster, you must include the file **BinaryInputBasic.h** in your application and enable the cluster by defining CLD_BINARY_INPUT_BASIC in the **zcl_options.h** file.

A Binary Input (Basic) cluster instance can act as either a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Binary Input (Basic) cluster are fully detailed in Section 19.3.6.

Parent topic: Binary Input (Basic) Cluster

Binary Input (Basic) Structure and Attributes The structure definition for the Binary Input (Basic) cluster is:

```
typedef struct
{
#ifdef BINARY_INPUT_BASIC_SERVER
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_ACTIVE_TEXT
    tsZCL_CharacterString       sActiveText;
    uint8                       au8ActiveText[16];
```

(continues on next page)

(continued from previous page)

```
#endif
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_DESCRIPTION
    tsZCL_CharacterString       sDescription;
    uint8                       au8Description[16];
#endif
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_INACTIVE_TEXT
    tsZCL_CharacterString       sInactiveText;
    uint8                       au8InactiveText[16];
#endif
    zbool                       bOutOfService;
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_POLARITY
    zenum8                      u8Polarity;
#endif
    zbool                       bPresentValue;
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_RELIABILITY
    zenum8                      u8Reliability;
#endif
    zbmap8                      u8StatusFlags;
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_APPLICATION_TYPE
    zuint32                     u32ApplicationType;
#endif
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8                      u8AttributeReportingStatus;
#endif
#endif
    zuint16                     u16ClusterRevision;
} tsCLD_BinaryInputBasic;
```

- The following optional pair of attributes are used to store a human readable description of the active state of a binary input (for example, “Window 3 open”):
  - sActiveText is a tsZCL_CharacterString structure (see Section6.1.14) for a string of up to 16 characters representing the description
  - au8ActiveText[16] is a byte-array which contains the character data bytes representing the description
    - If these attributes are used, the ‘Inactive Text’ attributes must also be used.
- The following optional pair of attributes are used to store a human readable description of the usage of the binary input (for example,”Window 3 status”):
  - sDescription is a tsZCL_CharacterString structure (see Section6.1.14) for a string of up to 16 characters representing the description
  - au8Description[16] is a byte-array which contains the character data bytes representing the description
- The following optional pair of attributes are used to store a human readable description of the inactive state of a binary input (for example,”Window 3 closed”):
  - sInactiveText is a tsZCL_CharacterString structure (see Section6.1.14) for a string of up to 16 characters representing the description
  - au8InactiveText[16] is a byte-array which contains the character data bytes representing the description
    - If these attributes are used, the ‘Active Text’ attributes must also be used.
- bOutOfService is a mandatory attribute which indicates whether the binary input is currently in or out of service:
  - TRUE: Out of service
  - FALSE In service

    * If this attribute is set to TRUE, the bPresentValue attribute will not be updated to contain the current value of the input.

- u8Polarity is an optional attribute which indicates the relationship between the value of the bPresentValue attribute and the physical state of the input:
  - E_CLD_ BINARY_INPUT_BASIC_POLARITY_NORMAL (0x00): The active (1) state of bPresentValue corresponds to the active/on state of the physical input
  - E_CLD_ BINARY_INPUT_BASIC_POLARITY_REVERSE (0x01): The active (1) state of bPresentValue corresponds to the inactive/off state of the physical input
- bPresentValue is a mandatory attribute representing the current state of the binary input (this attribute is updated when the input changes state):
  - TRUE: Input is in the 'active' state
  - FALSE: Input is in the 'inactive' state
    * The interpretation of bPresentValue in relation to the physical state of the input is determined by the setting of the u8Polarity attribute.
    * By default this attribute is read-only, but it becomes readable and writable when bOutOfService is set to TRUE.
- u8Reliability is an optional attribute which indicates whether the value reported through bPresentValue is reliable or why it might be unreliable:
  - E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED
  - E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_NO_SENSOR
  - E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_OVER_RANGE
  - E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_UNDER_RANGE
  - E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_OPEN_LOOP
  - E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_SHORTED_LOOP
  - E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_NO_OUTPUT
  - E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER
  - E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_PROCESS_ERROR
  - E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
- u8StatusFlags is a mandatory attribute which is a bitmap representing the following status flags:

Bits	Name	Description
0	In Alarm	Reserved - unused for Binary Input (Basic) cluster
1	Fault	- 1: Optional attribute u8Reliability is used and does not have a value of NO_FAULT_DETECTED

- 0: Otherwise

| |2|Overridden|- 1: Cluster has been over-ridden by a local mechanism

(bPresentValue and u8Reliability will not track input)

- 0: Otherwise

| |3|Out Of Service|- 1: Attribute bOutOfService is set to TRUE

- 0: Otherwise

| |4-7|-|Reserved|

- u32ApplicationType is an optional attribute which is a bitmap representing the application type, as follows:

Bits	Field Name	Description
0-15	Index	Specific application usage (for example, Boiler Status). There is a set of possible usages for each value of Type (see below). For these lists, refer to the attribute description in the ZCL Specification.
16-23	Type	Application domain. For the Basic Input cluster, this can be set to 0x00 (HVAC) or 0x01 (Security).
24-31	Group	Identifier for the cluster that this attribute is part of (not the Clus-ter ID). For the Binary Input (Basic) cluster, this is 0x03.

```
u8AttributeReportingStatus is an optional attribute that should be enabled when attribute report-
ing is used for the cluster (see [Section 2.3.5](../../ZCL_fundamentals/topics/attribute_reporting.md#id_
↪d3e2d682-d5e6-4aa5-870f-4d2e0320637e)). The value of this attribute indicates whether there are at-
tribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved.
↪ This attribute is also described in [Section 2.4](../../ZCL_fundamentals/topics/global_attributes.md#id_
↪8f80e769-8ccd-4b4f-8609-579f4457d527).
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specifica-
tion on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster re-
vision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This at-
tribute is also described in [Section 2.4](../../ZCL_fundamentals/topics/global_attributes.md#id_8f80e769-
↪8ccd-4b4f-8609-579f4457d527).
```

Parent topic: Binary Input (Basic) Cluster

Attributes for Default Reporting

The following attributes of the Binary Input (Basic) cluster can be selected for default reporting:

```
bPresentValue
u8AttributeReportingStatus
```

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic: Binary Input (Basic) Cluster

Functions

The following Binary Input (Basic) cluster function is provided in the NXP implementation of the ZCL:

- **Function** **Page**
- eCLD_BinaryInputBasicCreateBinaryInputBasic 432

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

eCLD_BinaryInputBasicCreateBinaryInputBasic

```
teZCL_Status eCLD_BinaryInputBasicCreateBinaryInputBasic(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
```

(continues on next page)

(continued from previous page)

```
uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Binary Input (Basic) cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Binary Input (Basic) cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Binary Input Basic cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Binary Input (Basic) cluster. This parameter can refer to a pre-filled structure called sCLD_BinaryInputBasic which is provided in the **BinaryInputBasic.h** file.
- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_BinaryInputBasic which defines the attributes of Binary Input (Basic) cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic: Binary Input (Basic) Cluster

Enumerations

teCLD_BinaryInputBasicCluster_AttrID The following structure contains the enumerations used to identify the attributes of the Binary Input (Basic) cluster.

```
typedef enum
{
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_ACTIVE_TEXT,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_DESCRIPTION,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_INACTIVE_TEXT,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_OUT_OF_SERVICE,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_POLARITY,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_PRESENT_VALUE,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_RELIABILITY,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_STATUS_FLAGS,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_APPLICATION_TYPE
} teCLD_BinaryInputBasicCluster_AttrID;
```

Parent topic: Enumerations

teCLD_BinaryInputBasic_Polarity The following structure contains the enumerations used to specify the value of the u8Polarity attribute (see Section 19.3.2).

```
typedef enum
{
   E_CLD_ BINARY_INPUT_BASIC_POLARITY_NORMAL                    ,
   E_CLD_ BINARY_INPUT_BASIC_POLARITY_REVERSE
}teCLD_BinaryInputBasic_Polarity
```

Parent topic: Enumerations

teCLD_BinaryInputBasic_Reliability The following structure contains the enumerations used to report the value of the u8Reliability attribute (see Section 19.3.2).

```
typedef enum
{
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_NO_SENSOR,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_OVER_RANGE,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_UNDER_RANGE,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_OPEN_LOOP,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_SHORTED_LOOP,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_NO_OUTPUT,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_PROCESS_ERROR,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
}teCLD_BinaryInputBasic_Reliability;
```

Parent topic: Enumerations

Parent topic: Binary Input (Basic) Cluster

Compile-time options To enable the Binary Input (Basic) cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_BINARY_INPUT_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define BINARY_INPUT_BASIC_CLIENT
#define BINARY_INPUT_BASIC_SERVER
```

Optional Attributes The optional attributes for the Binary Input (Basic) cluster (see Section 19.3.2) are enabled by defining:

- CLD_BINARY_INPUT_BASIC_ATTR_ACTIVE_TEXT
- CLD_BINARY_INPUT_BASIC_ATTR_DESCRIPTION
- CLD_BINARY_INPUT_BASIC_ATTR_INACTIVE_TEXT
- CLD_BINARY_INPUT_BASIC_ATTR_POLARITY
- CLD_BINARY_INPUT_BASIC_ATTR_RELIABILITY
- CLD_BINARY_INPUT_BASIC_ATTR_APPLICATION_TYPE

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_BINARY_INPUT_BASIC_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_BINARY_INPUT_BASIC_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic: Binary Input (Basic) Cluster

Parent topic:*Input and Output Clusters*

Binary Output (Basic) This chapter describes the Binary Output (Basic) cluster, which provides an interface for setting the state of a binary (two-state) output.

The Binary Output (Basic) cluster has a Cluster ID of 0x0010.

Overview The Binary Output (Basic) cluster provides an interface for setting the state of a binary output and its associated characteristics. It is typically used to implement a controller that produces a two-state output signal.

To use the functionality of this cluster, you must include the file **BinaryOutputBasic.h** in your application and enable the cluster by defining CLD_BINARY_OUTPUT_BASIC in the **zcl_options.h** file.

A Binary Output (Basic) cluster instance can act as either a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in

addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Binary Output (Basic) cluster are fully detailed in Section 19.4.6.

Parent topic: Binary Output (Basic)

Binary Output (Basic) Structure and Attributes The structure definition for the Binary Output (Basic) cluster is:

```
typedef struct
{
#ifdef BINARY_OUTPUT_BASIC_SERVER
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_ACTIVE_TEXT
    tsZCL_CharacterString       sActiveText;
    uint8                       au8ActiveText[16];
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_DESCRIPTION
    tsZCL_CharacterString       sDescription;
    uint8                       au8Description[16];
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_INACTIVE_TEXT
    tsZCL_CharacterString       sInactiveText;
    uint8                       au8InactiveText[16];
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_MINIMUM_OFF_TIME
    zuint32                     u32MinimumOffTime;
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_MINIMUM_ON_TIME
    zuint32                     u32MinimumOnTime;
#endif
    zbool                       bOutOfService;
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_POLARITY
    zenum8                      u8Polarity;
#endif
    zbool                       bPresentValue;
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_RELIABILITY
    zenum8                      u8Reliability;
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_RELINQUISH_DEFAULT
    zbool                       bRelinquishDefault;
#endif
    zbmap8                      u8StatusFlags;
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_APPLICATION_TYPE
    zuint32                     u32ApplicationType;
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8                      u8AttributeReportingStatus;
#endif
#endif
    zuint16                     u16ClusterRevision;
} tsCLD_BinaryOutputBasic;
```

- The following optional pair of attributes are used to store a human readable description of the active state of a binary output (e.g. “Open Window 3”):
  - sActiveText is a tsZCL_CharacterString structure (see Section6.1.14) for a string of up to 16 characters representing the description
  - au8ActiveText[16] is a byte-array which contains the character data bytes representing the description
    - If these attributes are used, the ‘Inactive Text’ attributes must also be used.

- The following optional pair of attributes are used to store a human readable description of the usage of the binary output (e.g. "Control Window 3"):
  - sDescription is a tsZCL_CharacterString structure (see Section6.1.14) for a string of up to 16 characters representing the description
  - au8Description[16] is a byte-array which contains the character data bytes representing the description
- The following optional pair of attributes are used to store a human readable description of the inactive state of a binary output (e.g. "Close Window 3"):
  - sInactiveText is a tsZCL_CharacterString structure (see Section6.1.14) for a string of up to 16 characters representing the description
  - au8InactiveText[16] is a byte-array which contains the character data bytes representing the description
    - If these attributes are used, the 'Active Text' attributes must also be used.

u32MinimumOffTime is an optional attribute which represents the minimum time, in seconds, for which the binary output will remain in the inactive state (0).

- u32MinimumOnTime is an optional attribute which represents the minimum time, in seconds, for which the binary output will remain in the active state (1).
- bOutOfService is a mandatory attribute which indicates whether the binary output is currently in or out of service:
  - TRUE: Out of service
  - FALSE: In service
    - If this attribute is set to TRUE, the bPresentValue attribute will not be used to control the binary output.
- u8Polarity is an optional attribute which indicates the relationship between the value of the bPresentValue attribute and the physical state of the output:
  - E_CLD_ BINARY_OUTPUT_BASIC_POLARITY_NORMAL (0x00): The active (1) state of bPresentValue corresponds to the active/on state of the physical output
  - E_CLD_ BINARY_OUTPUT_BASIC_POLARITY_REVERSE (0x01): The active (1) state of bPresentValue corresponds to the inactive/off state of the physical output
- bPresentValue is a mandatory attribute representing the current state of the binary output (this attribute is updated by the application):
  - TRUE: Output is in the 'active' state
  - FALSE: Output is in the 'inactive' state
    - The interpretation bPresentValue in relation to the physical state of the output is determined by the setting of the u8Polarity attribute.
- u8Reliability is an optional attribute which indicates whether the value contained in bPresentValue is reliable or why it might be unreliable:
  - E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED
  - E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_OVER_RANGE
  - E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_UNDER_RANGE
  - E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_OPEN_LOOP
  - E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_SHORTED_LOOP
  - E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER
  - E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_PROCESS_ERROR

  - E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR

fRelinquishDefault is an optional attribute representing the default value to be used for bPresentValue when the supplied value is invalid.

- u8StatusFlags is a mandatory attribute which is a bitmap representing the following status flags:

Bits	Name	Description
0	In Alarm	Reserved - unused for Binary Output (Basic) cluster
1	Fault	- 1: Optional attribute u8Reliability is used and does not have a value of NO_FAULT_DETECTED

- 0: Otherwise

| |2|Overridden|- 1: Cluster has been over-ridden by a local mechanism

(bPresentValue and u8Reliability will not track input)

- 0: Otherwise

| |3|Out Of Service|- 1: Attribute bOutOfService is set to TRUE

- 0: Otherwise

| |4-7|-|Reserved|

- u32ApplicationType is an optional attribute which is a bitmap representing the application type, as follows:

Bits	Field Name	Description
0-15	Index	Specific application usage (e.g. Heating Valve). There is a set of possible usages for each value of Type (see below). For these lists, refer to the attribute description in the ZCL Specification.
16-23	Type	Application domain. For the Basic Output cluster, this can be set to 0x00 (HVAC) or 0x01 (Security).
24-31	Group	Identifier for the cluster that this attribute is part of (not the Clus-ter ID). For the Binary Output (Basic) cluster, this is 0x04.

u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see [Section 2.3.5](../../ZCL_fundamentals/topics/attribute_reporting.md#id_d3e2d682-d5e6-4aa5-870f-4d2e0320637e)). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in [Section 2.4](../../ZCL_fundamentals/topics/global_attributes.md#id_8f80e769-8ccd-4b4f-8609-579f4457d527).
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in [Section 2.4](../../ZCL_fundamentals/topics/global_attributes.md#id_8f80e769-8ccd-4b4f-8609-579f4457d527).

Parent topic: Binary Output (Basic)

Attributes for Default Reporting The following attributes of the Binary Output (Basic) cluster can be selected for default reporting:

```
bPresentValue
u8AttributeReportingStatus
```

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic: Binary Output (Basic)

Functions The following Binary Output (Basic) cluster function is provided in the NXP implementation of the ZCL:

- – eCLD_BinaryOutputBasicCreateBinaryOutputBasic 442

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

eCLD_BinaryOutputBasicCreateBinaryOutputBasic

```
teZCL_Status eCLD_BinaryOutputBasicCreateBinaryOutputBasic(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Binary Output (Basic) cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Binary Output (Basic) cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Binary Output Basic cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client

- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Binary Output (Basic) cluster. This parameter can refer to a pre-filled structure called sCLD_BinaryOutputBasic which is provided in the **BinaryOutputBasic.h** file.
- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_BinaryOutputBasic which defines the attributes of Binary Output (Basic) cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic: Binary Output (Basic)

Enumerations

teCLD_BinaryOutputBasicCluster_AttrID The following structure contains the enumerations used to identify the attributes of the Binary Output (Basic) cluster.

```
typedef enum
{
   E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_ACTIVE_TEXT,
   E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_DESCRIPTION,
   E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_INACTIVE_TEXT,
   E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_MINIMUM_OFF_TIME,
   E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_MINIMUM_ON_TIME,
   E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_OUT_OF_SERVICE,
   E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_POLARITY,
   E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_PRESENT_VALUE,
   E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_RELIABILITY,
   E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_RELINQUISH_DEFAULT,
   E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_STATUS_FLAGS,
   E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_APPLICATION_TYPE,
} teCLD_BinaryOutputBasicCluster_AttrID;
```

Parent topic: Enumerations

teCLD_BinaryOutputBasic_Polarity The following structure contains the enumerations used to specify the value of the u8Polarity attribute (see Section 19.4.2).

```
typedef enum
{
   E_CLD_BINARY_OUTPUT_BASIC_POLARITY_NORMAL,
   E_CLD_BINARY_OUTPUT_BASIC_POLARITY_REVERSE
}teCLD_BinaryOutputBasic_Polarity
```

Parent topic: Enumerations

teCLD_BinaryOutputBasic_Reliability The following structure contains the enumerations used to report the value of the u8Reliability attribute (see Section 19.4.2).

```
typedef enum
{
   E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED,
   E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_OVER_RANGE,
   E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_UNDER_RANGE,
   E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_OPEN_LOOP,
   E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_SHORTED_LOOP,
   E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER,
   E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_PROCESS_ERROR,
   E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
 }teCLD_BinaryOutputBasic_Reliability;
```

Parent topic: Enumerations

Parent topic: Binary Output (Basic)

Compile-time options To enable the Binary Output (Basic) cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_BINARY_OUTPUT_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define BINARY_OUTPUT_BASIC_CLIENT
#define BINARY_OUTPUT_BASIC_SERVER
```

Optional Attributes The optional attributes for the Binary Output (Basic) cluster (see Section 19.4.2) are enabled by defining:

- CLD_BINARY_OUTPUT_BASIC_ATTR_ACTIVE_TEXT
- CLD_BINARY_OUTPUT_BASIC_ATTR_DESCRIPTION
- CLD_BINARY_OUTPUT_BASIC_ATTR_INACTIVE_TEXT
- CLD_BINARY_OUTPUT_BASIC_ATTR_MINIMUM_OFF_TIME
- CLD_BINARY_OUTPUT_BASIC_ATTR_MINIMUM_ON_TIME
- CLD_BINARY_OUTPUT_BASIC_ATTR_POLARITY
- CLD_BINARY_OUTPUT_BASIC_ATTR_RELIABILITY
- CLD_BINARY_OUTPUT_BASIC_ATTR_RELINQUISH_DEFAULT
- CLD_BINARY_OUTPUT_BASIC_ATTR_APPLICATION_TYPE
- CLD_BINARY_OUTPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS

Parent topic: Binary Output (Basic)

Parent topic:*Input and Output Clusters*

Multistate Input (Basic) This chapter describes the Multistate Input (Basic) cluster, which provides an interface for accessing a multistate measurement (that can take one of a set of fixed states).

The Multistate Input (Basic) cluster has a Cluster ID of 0x0012.

Overview The Multistate Input (Basic) cluster provides an interface for accessing a multistate measurement and its associated characteristics. It is typically used in a sensor that measures a physical quantity that can take one of a discrete number of states.

To use the functionality of this cluster, you must include the file **MultistateInputBasic.h** in your application and enable the cluster by defining CLD_MULTISTATE_INPUT_BASIC in the **zcl_options.h** file.

A Multistate Input (Basic) cluster instance can act as either a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Multistate Input (Basic) cluster are fully detailed in Section 19.5.6.

Parent topic: Multistate Input (Basic)

Multistate Input (Basic) Structure and Attributes The structure definition for the Multistate Input (Basic) cluster is:

```
typedef struct
{
#ifdef MULTISTATE_INPUT_BASIC_SERVER
#ifdef CLD_MULTISTATE_INPUT_BASIC_ATTR_DESCRIPTION
    tsZCL_CharacterString       sDescription;
    uint8                       au8Description[16];
#endif
    zuint16                     u16NumberOfStates;
    zbool                       bOutOfService;
    zuint16                     u16PresentValue;
#ifdef CLD_MULTISTATE_INPUT_BASIC_ATTR_RELIABILITY
    zenum8                      u8Reliability;
#endif
    zbmap8                      u8StatusFlags;
#ifdef CLD_MULTISTATE_INPUT_BASIC_ATTR_APPLICATION_TYPE
    zuint32                     u32ApplicationType;
#endif
#ifdef CLD_MULTISTATE_INPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8                      u8AttributeReportingStatus;
#endif
#endif
    zuint16                     u16ClusterRevision;
} tsCLD_MultistateInputBasic;
```

- The following optional pair of attributes are used to store a human readable description of the usage of the multistate input (e.g. "Alarm Status"):
  - sDescription is a tsZCL_CharacterString structure (see Section6.1.14) for a string of up to 16 characters representing the description
  - au8Description[16] is a byte-array which contains the character data bytes representing the description

u16NumberOfStates is a mandatory attribute which indicates the number of discrete states that the input can take.

- bOutOfService is a mandatory attribute which indicates whether the multistate input is currently in or out of service:
  - TRUE: Out of service
  - FALSE In service
    - If this attribute is set to TRUE, the u16PresentValue attribute is not updated to contain the current state of the input.
- u16PresentValue is a mandatory attribute representing the latest state of the input (this attribute is updated when the multistate input changes).
  - By default this attribute is read-only, but it becomes readable and writable when bOutOfService is set to TRUE.
- u8Reliability is an optional attribute which indicates whether the value reported through fPresentValue is reliable or why it might be unreliable:
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_NO_SENSOR
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_OVER_RANGE
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_UNDER_RANGE
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_OPEN_LOOP
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_SHORTED_LOOP
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_NO_OUTPUT
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_PROCESS_ERROR
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_MULTISTATE_FAULT
- u8StatusFlags is a mandatory attribute which is a bitmap representing the following status flags:

Bits	Name	Description
0	In Alarm	Reserved - unused for Multistate Input (Basic) cluster
1	Fault	- 1: Optional attribute u8Reliability is used and does not have a value of NO_FAULT_DETECTED

- 0: Otherwise

| |2|Overridden|- 1: Cluster has been over-ridden by a local mechanism

(fP``resentValue and u8Reliability will not track input)

- 0: Otherwise

| |3|Out Of Service|- 1: Attribute bOutOfService is set to TRUE

- 0: Otherwise

| |4-7|-|Reserved|

- u32ApplicationType is an optional attribute that is a bitmap representing the application type, as follows:

Bits	Field Name	Description
0-15	Index	Specific application usage in terms of the states supported

(e.g. Off/On/Auto). For the list of usages, refer to the attribute description in the ZCL Specification.

| |16-23|Type|Application domain. For the Multistate Input cluster, this can only be is set to 0x00 (HVAC).| |24-31|Group|Identifier for the cluster that this attribute is part of (not the Cluster ID). For the Multistate Input cluster, this is 0x0D.|

- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic: Multistate Input (Basic)

Attributes for Default Reporting The following attributes of the Multistate Input (Basic) cluster can be selected for default reporting:

```
u16PresentValue
u8AttributeReportingStatus
```

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic: Multistate Input (Basic)

Functions The following Multistate Input (Basic) cluster function is provided in the NXP implementation of the ZCL:

- – eCLD_MultistateInputBasicCreateMultistateInputBasic 450

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

eCLD_MultistateInputBasicCreateMultistateInputBasic

```
teZCL_Status eCLD_MultistateInputBasicCreateMultistateInputBasic(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Multistate Input (Basic) cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Multistate Input (Basic) cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Multistate Input Basic cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Multistate Input (Basic) cluster. This parameter can refer to a pre-filled structure called sCLD_MultistateInputBasic which is provided in the **MultistateInputBasic.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_MultistateInputBasic which defines the attributes of Multistate Input (Basic) cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic: Multistate Input (Basic)

Enumerations

teCLD_MultistateInputBasicCluster_AttrID The following structure contains the enumerations used to identify the attributes of the Multistate Input (Basic) cluster.

```
typedef enum
{
    E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_DESCRIPTION,
    E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_NUMBER_OF_STATES,
    E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_OUT_OF_SERVICE,
```

(continues on next page)

(continued from previous page)

```
    E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_PRESENT_VALUE,
    E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_RELIABILITY,
    E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_STATUS_FLAGS,
    E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_APPLICATION_TYPE,
}teCLD_MultistateInputBasicCluster_AttrID;
```

Parent topic: Enumerations

teCLD_MultistateInputBasic_Reliability The following structure contains the enumerations used to report the value of the u8Reliability attribute (see Section 19.5.2).

```
typedef enum
{
  E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED,
  E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_NO_SENSOR,
  E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_OVER_RANGE,
  E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_UNDER_RANGE,
  E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_OPEN_LOOP,
  E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_SHORTED_LOOP,
  E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_NO_OUTPUT,
  E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER,
  E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_PROCESS_ERROR,
  E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_MULTISTATE_FAULT,
  E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
}teCLD_MultistateInputBasic_Reliability;
```

Parent topic: Enumerations

Parent topic: Multistate Input (Basic)

Compile-time options To enable the Multistate Input (Basic) cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_MULTISTATE_INPUT_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define MULTISTATE_INPUT_BASIC_CLIENT
#define MULTISTATE_INPUT_BASIC_SERVER
```

Optional Attributes The optional attributes for the Multistate Input (Basic) cluster (see Section 19.5.2) are enabled by defining:

- CLD_MULTISTATE_INPUT_BASIC_ATTR_DESCRIPTION
- CLD_MULTISTATE_INPUT_BASIC_ATTR_RELIABILITY
- CLD_MULTISTATE_INPUT_BASIC_ATTR_APPLICATION_TYPE
- CLD_MULTISTATE_INPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_MULTISTATE_INPUT_BASIC_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic: Multistate Input (Basic)

Parent topic:*Input and Output Clusters*

Multistate Output (Basic) This chapter describes the Multistate Output (Basic) cluster, which provides an interface for setting the value of a multistate output (that can take one of a set of fixed states).

The Multistate Output (Basic) cluster has a Cluster ID of 0x0013.

Overview The Multistate Input (Basic) cluster provides an interface for setting the value of a multistate output and its associated characteristics. It is typically used in a controller which outputs a control signal that can be set to one of a discrete number of states.

To use the functionality of this cluster, you must include the file **MultistateOutputBasic.h** in your application and enable the cluster by defining CLD_MULTISTATE_OUTPUT_BASIC in the **zcl_options.h** file.

An Multistate Output (Basic) cluster instance can act as either a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Multistate Output (Basic) cluster are fully detailed in Section 19.6.6.

Parent topic: Multistate Output (Basic)

Multistate Output (Basic) Structure and Attributes The structure definition for the Multistate Output (Basic) cluster is:

```
typedef struct
{
#ifdef MULTISTATE_OUTPUT_BASIC_SERVER
#ifdef CLD_MULTISTATE_OUTPUT_BASIC_ATTR_DESCRIPTION
    tsZCL_CharacterString       sDescription;
    uint8                       au8Description[16];
#endif
    zuint16                     u16NumberOfStates;
    zbool                       bOutOfService;
    zuint16                     u16PresentValue;
#ifdef CLD_MULTISTATE_OUTPUT_BASIC_ATTR_RELIABILITY
    zenum8                      u8Reliability;
#endif
#ifdef CLD_MULTISTATE_OUTPUT_BASIC_ATTR_RELINQUISH_DEFAULT
    zuint16                     u16RelinquishDefault;
#endif
```

(continues on next page)

(continued from previous page)

```
    zbmap8                  u8StatusFlags;
#ifdef CLD_MULTISTATE_OUTPUT_BASIC_ATTR_APPLICATION_TYPE
    zuint32                 u32ApplicationType;
#endif
#ifdef CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8                  u8AttributeReportingStatus;
#endif
#endif
    zuint16             u16ClusterRevision;
} tsCLD_MultistateOutputBasic;
```

- The following optional pairs of attributes are used to store a human readable description of the usage of the multistate output (for example "Alarm State"):
  - sDescription is a tsZCL_CharacterString structure (see Section 6.1.14) for a string of up to 16 characters representing the description
  - au8Description[16] is a byte-array which contains the character data bytes representing the description.
- u16NumberOfStates is a mandatory attribute that indicates the number of discrete states that the output can take.
- bOutOfService is a mandatory attribute that indicates whether the multistate output is in or out of service currently:
  - TRUE: Out of service
  - FALSE: In service
    - If this attribute is set to TRUE, the u16PresentValue attribute is not used to control the multistate output.
- u16PresentValue is a mandatory attribute representing the latest multistate output value (this attribute is used to control the physical output).
- u8Reliability is an optional attribute that indicates whether the value contained in u16PresentValue is reliable or why it might be unreliable:
  - E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED
  - E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_OVER_RANGE
  - E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_UNDER_RANGE
  - E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_OPEN_LOOP
  - E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_SHORTED_LOOP
  - E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER
  - E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_PROCESS_ERROR
  - E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_MULTISTATE_FAULT
  - E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
- fRelinquishDefault is an optional attribute representing the default value to be used for u16PresentValue when the supplied value is invalid.
- u8StatusFlags is a mandatory attribute, which is a bitmap representing the following status flags:

Bits	Name	Description
0	In Alarm	Reserved - unused for Multistate Output (Basic) cluster
1	Fault	- 1: Optional attribute u8Reliability is used and does not have a value of NO_FAULT_DETECTED

- 0: Otherwise

| |2|Overridden|- 1: Cluster has been over-ridden by a local mechanism

(u16P``resentValue and u8Reliability do not track input)

- 0: Otherwise

| |3|Out Of Service|- 1: Attribute bOutOfService is set to TRUE

- 0: Otherwise

| |4-7|-|Reserved|

- u32ApplicationType is an optional attribute which is a bitmap representing the application type, as follows:

Bits	Field Name	Description
0-15	Index	Specific application usage in terms of the states supported

(for example, Off/On/Auto). For the list of usages, refer to the attribute description in the ZCL Specification.

| |16-23|Type|Application domain. For the Multistate Output cluster, this field can only be set to 0x00 (HVAC).| |24-31|Group|Identifier for the cluster that this attribute is part of (not the Cluster ID). For the Multistate Output cluster, this is 0x0E.|

- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic: Multistate Output (Basic)

Attributes for Default Reporting The following attributes of the Multistate Output (Basic) cluster can be selected for default reporting:

```
u16PresentValue
u8AttributeReportingStatus
```

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic: Multistate Output (Basic)

Functions The following Multistate Output (Basic) cluster function is provided in the NXP implementation of the ZCL:

- eCLD_MultistateOutputBasicCreateMultistateOutputBasic

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

eCLD_MultistateOutputBasicCreateMultistateOutputBasic

```
teZCL_Status eCLD_MultistateOutputBasicCreateMultistateOutputBasic(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    int8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Multistate Output (Basic) cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create an Multistate Output (Basic) cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Multistate Output Basic cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Multistate Output (Basic) cluster. This parameter can refer to a pre-filled structure called sCLD_MultistateOutputBasic which is provided in the **MultistateOutputBasic.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_MultistateOutputBasic which defines the attributes of Multistate Output (Basic) cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic: Multistate Output (Basic)

Enumerations

teCLD_MultistateOutputBasicCluster_AttrID The following structure contains the enumerations used to identify the attributes of the Multistate Output (Basic) cluster.

```
typedef enum {
  E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_DESCRIPTION,
  E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_NUMBER_OF_STATES,
  E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_OUT_OF_SERVICE,
  E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_PRESENT_VALUE,
  E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_RELIABILITY,
  E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_RELINQUISH_DEFAULT,
  E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_STATUS_FLAGS,
  E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_APPLICATION_TYPE,
} teCLD_MultistateOutputBasicCluster_AttrID;
```

Parent topic: Enumerations

teCLD_MultistateOutputBasic_Reliability The following structure contains the enumerations used to report the value of the u8Reliability attribute (see Section 19.6.2).

```
typedef enum
{
  E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED,
  E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_OVER_RANGE,
  E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_UNDER_RANGE,
  E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_OPEN_LOOP,
  E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_SHORTED_LOOP,
  E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER,
  E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_PROCESS_ERROR,
  E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_MULTISTATE_FAULT,
  E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
}teCLD_MultistateOutputBasic_Reliability;
```

Parent topic: Enumerations

Parent topic: Multistate Output (Basic)

Compile-time options To enable the Multistate Output (Basic) cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_MULTISTATE_OUTPUT_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define MULTISTATE_OUTPUT_BASIC_CLIENT
#define MULTISTATE_OUTPUT_BASIC_SERVER
```

Optional attributes The optional attributes for the Multistate Output (Basic) cluster (see Section 19.6.2) are enabled by defining:

- CLD_MULTISTATE_OUTPUT_BASIC_ATTR_DESCRIPTION
- CLD_MULTISTATE_OUTPUT_BASIC_ATTR_RELIABILITY
- CLD_MULTISTATE_OUTPUT_BASIC_ATTR_RELINQUISH_DEFAULT
- CLD_MULTISTATE_OUTPUT_BASIC_ATTR_APPLICATION_TYPE
- CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS

Global attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_MULTISTATE_OUTPUT_BASIC_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic: Multistate Output (Basic)

Parent topic:*Input and Output Clusters*

Poll Control Cluster This chapter describes the Poll Control cluster which provides an interface for remotely controlling the rate at which a ZigBee End Device polls its parent for data.

The Poll Control cluster has a Cluster ID of 0x0020.

Overview An End Device cannot receive data packets directly, as it might be asleep when a packet arrives. The data packets for an End Device are therefore buffered by the device's parent and the End Device polls its parent for data while awake. An individual data packet is only held on the parent node for a maximum of 7.68 seconds and if many packets for the End Device are expected over a short period of time, the End Device should retrieve these packets as quickly as possible. An End Device can implement two polling modes, which are dependent on the poll interval (time-period between consecutive polls):

- **Normal poll mode:** A long poll interval is used - this mode is appropriate when the End Device is not expecting data packets.
- **Fast poll mode:** A short poll interval is used - this mode is appropriate when the End Device is expecting data packets.

The End Device may enable fast poll mode itself when it is expecting data packets (for example, after it has requested data from remote nodes). The Poll Control cluster allows fast poll mode to be selected from a remote control device to force the End Device to be more receptive to data packets (for example, when a download to the End Device involving a large number of unsolicited data packets is to be initiated).

The two sides of the cluster are located as follows:

- The cluster server is implemented on the End Device to be controlled
- The cluster client is implemented on the remote controller device

The cluster server (End Device) periodically checks whether the cluster client (remote controller) requires the poll mode to be changed. This 'check-in' method is used since an unsolicited instruction from the controller may arrive when the End Device is asleep. The automatic 'check-ins' are conducted with all the remote endpoints (on controller nodes) to which the local endpoint (on which the cluster resides) is bound.

The cluster is enabled by defining CLD_POLL_CONTROL in the **zcl_options.h** file. Further compile-time options for the Poll Control cluster are detailed in Section 20.10.

Parent topic:*Poll Control Cluster*

Cluster structure and attributes The structure definition for the Poll Control cluster (server) is:

```
typedef struct
{
#ifdef POLL_CONTROL_SERVER
    zuint32     u32CheckinInterval;
    zuint32     u32LongPollInterval;
    zuint16     u16ShortPollInterval;
    zuint16     u16FastPollTimeout;
#ifdef CLD_POLL_CONTROL_ATTR_CHECKIN_INTERVAL_MIN
    zuint32     u32CheckinIntervalMin;
#endif
#ifdef CLD_POLL_CONTROL_ATTR_LONG_POLL_INTERVAL_MIN
    zuint32     u32LongPollIntervalMin;
#endif
#ifdef CLD_POLL_CONTROL_ATTR_FAST_POLL_TIMEOUT_MAX
    zuint16     u16FastPollTimeoutMax;
#endif
#endif
    zuint16     u16ClusterRevision;
} tsCLD_PollControl;
```

where:

- u32CheckinInterval is the 'check-in interval', used by the server in checking whether a client requires the poll mode to be changed - this is the period, in quarter-seconds, between consecutive checks. The valid range of values is 1 to 7208960. A user-defined minimum value for this attribute can be set via the optional attribute u32CheckinIntervalMin (see below). Zero is a special value indicating that the Poll Control cluster server is disabled. The default value is 14400 (1 hour).
- u32LongPollInterval is the 'long poll interval' of the End Device, employed when operating in normal poll mode - this is the period, in quarter-seconds, between consecutive polls of the parent for data. The valid range of values is 4 to 7208960. A user-defined minimum value for this attribute can be set via the optional attribute u32LongPollIntervalMin (see below). 0xFFFF is a special value indicating that the long poll interval is unknown/undefined. The default value is 20 (5 seconds).
- u16ShortPollInterval is the 'short poll interval' of the End Device, employed when operating in fast poll mode - this is the period, in quarter-seconds, between consecutive polls of the parent for data. The valid range of values is 1 to 65535 and the default value is 2 (0.5 seconds).
- u16FastPollTimeout is the 'fast poll timeout' representing the time-interval, in quarter-seconds, for which the server should normally stay in fast poll mode (unless over-ridden

by a client command). The valid range of values is 1 to 65535. It is recommended that this timeout is greater than 7.68 seconds. A user-defined maximum value for this attribute can be set via the optional attribute u16FastPollTimeoutMax (see below). The default value is 40

(10 seconds).

- u32CheckinIntervalMin is an optional lower limit on the 'check-in interval' defined by u32CheckinInterval. This limit can be used to ensure that the interval is not inadvertently set to a low value which will quickly drain the energy resources of the End Device node.
- u32LongPollIntervalMin is an optional lower limit on the 'long poll interval' defined by u32LongPollInterval. This limit can be used to ensure that the interval is not inadvertently set (for example, by another device) to a low value which will quickly drain the energy resources of the End Device node.
- u16FastPollTimeoutMax is an optional upper limit on the 'fast poll timeout' defined by u16FastPollTimeout. This limit can be used to ensure that the interval is not inadvertently set (for example, by another device) to a high value which quickly drains the energy resources of the End Device node.

Note:

1. Valid ranges (maximum and minimum values) for the four mandatory attributes can alternatively be set using macros in the **zcl_options.h** file, as described in Section 20.10. Some of these macros can only be used when the equivalent optional attribute is disabled.
2. For general guidance on attribute settings, refer to Section 20.3. Configuration through the attributes is also described in Section 20.4.2.

u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Poll Control Cluster*

Attribute Settings In assigning user-defined values to the mandatory attributes, the following inequality should be obeyed:

u32CheckinInterval u32LongPollInterval u16ShortPollInterval

In addition, the mandatory attribute u16FastPollTimeout should not be set to an excessive value for self-powered nodes, as fast poll mode can rapidly drain the stored energy of a node (for example, the battery).

The three optional attributes can be used to ensure that the values of the corresponding mandatory attributes are kept within reasonable limits, to prevent the rapid depletion of the energy resources of the node. If required, the optional attributes must be enabled and initialized in the compile-time options (see Section 20.10).

Minimum and maximum values for all the mandatory attributes can alternatively be set using the compile-time options (again, refer to Section 20.10).

Parent topic:*Poll Control Cluster*

Poll Control Operations This section describes the main operations to be performed on the Poll Control cluster server (End Device) and client (controller).

Initialization The Poll Control cluster must be initialized on both the cluster server and client. This can be done using the function **eCLD_PollControlCreatePollControl()**, which creates an instance of the Poll Control cluster on a local endpoint.

If you are using a standard ZigBee device which includes the Poll Control cluster, the above function is automatically called by the initialization function for the device. You only need to call **eCLD_PollControlCreatePollControl()** explicitly when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device).

Parent topic: Poll Control Operations

Configuration When initialized, the Poll Control cluster adopts the attribute values that are pre-set in the tsCLD_PollControl structure (see Section 20.2). For the optional attributes, values can be set in the file **zcl_options.h** (see Section 20.10).

The mandatory attributes (and related optional attributes) are as follows:

- **Long Poll Interval (****u32LongPollInterval****)**: This is the polling period used in normal poll mode, expressed in quarter-seconds, with a default value of 20 (5 seconds). The attribute has a valid range of 4 to 7208960 but a user-defined minimum value for this attribute can be set via the optional 'long poll interval maximum' attribute (u32LongPollIntervalMin). This limit can be used to ensure that the interval is not inadvertently set (for example, by another device) to a low value which quickly drains the energy resources of the End Device node. Alternatively, minimum and maximum values can be specified through the compile-time options (see Section 20.10).
- **Short Poll Interval (****u16ShortPollInterval****)**: This is the polling period used in fast poll mode, expressed in quarter-seconds, with a default value of 2 (0.5 seconds). The attribute has a valid range of 1 to 65535. User-defined minimum and maximum values for this attribute can be specified through the compile-time options (see Section 20.10).
- **Fast Poll Timeout (****u16FastPollTimeout****)**: This is the time-interval for which the server should normally stay in fast poll mode (unless over-ridden by a client command), expressed in quarter-seconds, with a default value of 40 (10 seconds). It is recommended that this timeout is greater than 7.68 seconds. The valid range of values is 1 to 65535 but a user-defined maximum value for this attribute can be set via the optional 'fast poll timeout maximum' attribute (u16FastPollTimeoutMax). This limit can be used to ensure that the interval is not inadvertently set (for example, by another device) to a high value which quickly drains the energy resources of the End Device node. Alternatively, minimum and maximum values can be specified through the compile-time options (see Section 20.10).
- **Check-in Interval (****u32CheckinInterval****)**: This is the period between the server's checks of whether a client requires the poll mode to be changed, expressed in quarter-seconds, with a default value of 14400 (1 hour). It should be greater than the 'long poll interval' (see above). Zero is a special value indicating that the Poll Control cluster server is disabled. Otherwise, the valid range of values is 1 to 7208960 but a user-defined minimum value for this attribute can be set via the optional 'check-in interval minimum' attribute (u32CheckinIntervalMin). This limit can be used to ensure that the interval is not inadvertently set to a low value which quickly drains the energy resources of the End Device node. Alternatively, minimum and maximum values can be specified through the compile-time options (see Section 20.10).

The Poll Control cluster server can also be configured by the server application at run-time by writing to the relevant attribute(s) using the **eCLD_PollControlSetAttribute()** function (which must be called separately for each attribute to be modified). If used, this function must be called after the cluster has been initialized (see Section 20.4.1).

Changes to certain attributes can also be initiated remotely from the cluster client (controller) using the following functions:

- **eCLD_PollControlSetLongPollIntervalSend()**: The client application can use this function to submit a request to set the 'long poll interval' attribute on the server to a specified value. This function causes a 'Set Long Poll Interval' command to be sent to the relevant End Device. If the new value is acceptable, the cluster server automatically updates the attribute.

- **eCLD_PollControlSetShortPollIntervalSend()**: The client application can use this function to submit a request to set the 'short poll interval' attribute on the server to a specified value. This function causes a 'Set Short Poll Interval' command to be sent to the relevant End Device. If the new value is acceptable, the cluster server automatically updates the attribute.

In both of the above cases, a response is only sent back to the client if the new value is not acceptable, in which case a ZCL 'default response' is sent indicating an invalid value.

Use of the above two functions requires the corresponding commands to be enabled in the compile-time options, as described in Section 20.10.

Note: Changes to attribute values initiated by either the server application or client application takes effect immediately. So, for example, if the End Device is operating in fast poll mode when the 'short poll interval' is modified, the polling period is immediately re-timed to the new value. If the modified attribute is not related to the currently operating poll mode, the change is implemented the next time the relevant poll mode is started.

Before the first scheduled 'check-in' (after one hour, by default), the End Device application should set up bindings between the local endpoint on which the cluster resides and the relevant endpoint on each remote controller node with which the End Device operates. These bindings are used while sending the 'Check-in' commands.

Parent topic: Poll Control Operations

Operation After initialization, the Poll Control cluster server on the End Device begins to operate in normal poll mode and performs the following activities (while the End Device is awake):

- Periodically poll the parent for data packets at a rate determined by the 'long poll interval'.
- Periodically check whether any bound cluster clients require the server to enter fast poll mode, with 'check-ins' at a rate determined by the 'check-in interval'.

The server application must provide the cluster with timing prompts for the above periodic activities. These prompts are produced by periodically calling the function **eCLD_PollControlUpdate()**. The periods of the above activities are defined in terms of quarter-seconds. Therefore, this function must be called every quarter-second and the application must provide a 250 ms software timer to schedule these calls. Any poll or check-in that is due when this function is called is automatically performed by the cluster server.

The End Device operates in normal poll mode until either it puts itself into fast poll mode (for example, when it is expecting responses to a request) or the controller (client) requests the End Device to enter fast poll mode (for example, when a data download to the End Device is going to be performed). As indicated above, such a request from the client is raised as the result of the server performing periodic 'check-ins' with the client.

On receiving a 'check-in' command, an E_CLD_POLL_CONTROL_CMD_CHECK_IN event is generated on the client. The client application must then fill in the tsCLD_PollControl_CheckinResponsePayload structure (see Section 20.9.2) of the event, indicating whether fast poll mode is required. A response is then automatically sent back to the server.

After sending the initial Check-in command, the server waits for up to 7.68 seconds for a response (if no response is received in this time, the server is free to continue in normal poll mode). If a response is received from a client, the event E_CLD_POLL_CONTROL_CMD_CHECK_IN is generated on the server, where this event indicates the processing status of the received response. The server also sends this status back to the responding client in a ZCL default response.

- If the response was received from a bound client within the timeout period of the initial Check-in command, the status is ZCL_SUCCESS. In this case, the End Device is automatically put into fast poll mode.
- If the response is invalid for some reason, an error status is indicated as described below in Section 20.4.3.2, and fast poll mode is not entered.

When the End Device is in fast poll mode, the client application can request the cluster server to exit fast poll mode immediately (before the timeout expires) by calling the function **eCLD_PollControlFastPollStopSend()**.

Fast Poll Mode Timeout In the Check-in response from a client, the payload (see Section 20.9.2) may contain an optional timeout value which, if used, specifies the length of time that the device should remain in fast poll mode (this timeout value will be used instead of the one specified through the 'fast poll timeout' attribute). If the response payload specifies an out-of-range timeout value, the server will send a ZCL default response with status INVALID_VALUE to the client (see Section 20.4.3.2). In the case of multiple clients (controllers) that have specified different timeout values, the server will use the largest timeout value received.

Parent topic: Operation

Invalid Check-in Responses The server may receive Check-in responses which cannot result in fast poll mode. In these cases, the server sends a ZCL default response indicating the relevant error status (which is not ZCL_SUCCESS) back to the originating client. The following circumstances will lead to such a default response:

- The Check-in response is from an unbound client. In this case, the Default Response will contain the status ACTION_DENIED.
- The Check-in response is from a bound client but requests an invalid fast poll timeout value (see Section 20.4.3.1). In this case, the default response will contain the status INVALID_VALUE.
- The Check-in response is from a bound client but arrives after the timeout period of the original Check-in command. In this case, the default response will contain the status TIMEOUT.

Parent topic: Operation

Parent topic: Poll Control Operations

Parent topic:*Poll Control Cluster*

Poll Control Events The Poll Control cluster has its own events that are handled through the callback mechanism described in *Chapter 3*. The cluster contains its own event handler. However, if a device uses this cluster then application-specific Poll Control event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function will then be invoked when a Poll Control event occurs and needs the attention of the application.

For a Poll Control event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_PollControlCallBackMessage structure:

```
typedef struct
{
 uint8 u8CommandId;
 union
 {
 tsCLD_PollControl_CheckinResponsePayload *psCheckinResponsePayload;
 #ifdef CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL
 tsCLD_PollControl_SetLongPollIntervalPayload
          *psSetLongPollIntervalPayload;
 #endif
 #ifdef CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL
      tsCLD_PollControl_SetShortPollIntervalPayload
```

(continues on next page)

(continued from previous page)

```
    *psSetShortPollIntervalPayload;
 #endif
 } uMessage;
} tsCLD_PollControlCallBackMessage;
```

The above structure is fully described in Section 20.9.1.

When a Poll Control event occurs, one of the command types listed in *Table 29* is specified through the u8CommandId field of the structure tsCLD_PollControlCallBackMessage. This command type determines which command payload is used from the union uMessage.

u8CommandId Enumeration	Description/Payload Type
On Client	
E_CLD_POLL_CONTROL_CM	A Check-in command has been received by the client.
On Server	
E_CLD_POLL_CONTROL_CM	A Check-in Response has been received by the server, following a previously sent Check-In command.

tsCLD_PollControl_CheckinResponsePayload

| |E_CLD_POLL_CONTROL_CMD_FAST_POLL_STOP|A 'Fast Poll Stop' command has been received by the server.| |E_CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL|A 'Set Long Poll Interval' command has been received by the server.

tsCLD_PollControl_SetLongPollIntervalPayload

| |E_CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL|A 'Set Short Poll Interval' command has been received by the server.

tsCLD_PollControl_SetShortPollIntervalPayload

|

Parent topic:*Poll Control Cluster*

Functions The Poll Control cluster functions are described in the following three sub-sections, according to the side(s) of the cluster on which they can be used:

- Server/client functions are described in Section 20.6.1
- Server functions are described in Section 20.6.2
- Client functions are described in Section 20.6.3

Server/Client Function The following Poll Control cluster function can be used on either a cluster server or cluster client:

- eCLD_PollControlCreatePollControl

eCLD_PollControlCreatePollControl

```
teZCL_Status eCLD_PollControlCreatePollControl(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
```

(continues on next page)

(continued from previous page)

```
void *pvEndPointSharedStructPtr,
uint8 *pu8AttributeControlBits,
tsCLD_PollControlCustomDataStructure *psCustomDataStructure);
```

Description This function creates an instance of the Poll Control cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Poll Control cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to *Appendix D.*

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in the *ZigBee Devices User Guide* (JNUG3131).

When used, this function must be the first Poll Control cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length is automatically adjusted by the compiler using the following declaration:

```
uint8 au8PollControlAttributeControlBits
[(sizeof(asCLD_PollControlClusterAttrDefs) / sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Poll Control cluster. This parameter can refer to a pre-filled structure called sCLD_PollControl which is provided in the **PollControl.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_PollControl which defines the attributes of the Poll Control cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 20.9.5).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Server/Client Function

Parent topic: Functions

Server Functions The following Poll Control cluster functions can be used on a cluster server only:

- eCLD_PollControlUpdate
- eCLD_PollControlSetAttribute
- eCLD_PollControlUpdateSleepInterval

eCLD_PollControlUpdate

```
teZCL_Status eCLD_PollControlUpdate(void);
```

Description This function can be used on a cluster server to update the timing status for the following periodic activities:

- polling of the parent for a data packet
- 'check-ins' with the client to check for a required change in the poll mode

The function should be called once per quarter-second and the application should provide a 250-ms timer to prompt these function calls.

Any poll or check-in that is due when this function is called are automatically performed by the cluster server.

Parameters

- None

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Server Functions

eCLD_PollControlSetAttribute

```
teZCL_Status eCLD_PollControlSetAttribute(
    uint8 u8SourceEndPointId,
    uint8 u8AttributeId,
    uint32 u32AttributeValue);*
```

Description This function can be used on a cluster server to write to an attribute of the Poll Control cluster. The function writes to the relevant field of the tsCLD_PollControlstructure (detailed in Section 20.2). The attribute to be accessed is specified using its attribute identifier - enumerations are provided (see Section 20.8.1).

Therefore, this function can be used to change the configuration of the Poll Control cluster. The change takes effect immediately. So, for example, if the End Device is in normal poll mode when the 'long poll interval' is modified, the polling period is immediately re-timed to the new value. If the modified attribute is not related to the currently operating poll mode, the change is implemented the next time the relevant poll mode is started.

The specified value of the attribute is validated by the function. If this value is out-of-range for the attribute, the status E_ZCL_ERR_INVALID_VALUE is returned.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster resides
- *u8AttributeId* : Identifier of attribute to be written to (see Section 20.8.1)
- *u32AttributeValue*: Value to be written to attribute

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_INVALID_VALUE
- E_ZCL_DENY_ATTRIBUTE_ACCESS

Parent topic: Server Functions

eCLD_PollControlUpdateSleepInterval

```
teZCL_Status eCLD_PollControlUpdateSleepInterval(
    uint32 u32QuarterSecondsAsleep);
```

Description This function can be used on a cluster server to provide the updated ticks back into PollControl cluster for the time the device was sleeping in terms of quarter second.

This function updates the Checkin period based on the ticks provided.

Parameters

- *u32QuarterSecondsAsleep*: Number of Quarter seconds the device has slept for

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_CLUSTER_NOT_FOUND

Parent topic: Server Functions

Parent topic: Functions

Client Functions The following Poll Control cluster functions can be used on a cluster client only:

1. eCLD_PollControlSetLongPollIntervalSend
2. eCLD_PollControlSetShortPollIntervalSend
3. eCLD_PollControlFastPollStopSend

eCLD_PollControlSetLongPollIntervalSend

```
teZCL_Status eCLD_PollControlSetLongPollIntervalSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PollControl_SetLongPollIntervalPayload
    *psPayload);
```

Description This function can be used on a cluster client to send a 'Set Long Poll Interval' command to the cluster server. This command requests the 'long poll interval' for normal poll mode on the End Device to be set to the specified value.

On receiving the command, the 'long poll interval' attribute is only modified by the server if the specified value is within the valid range for the attribute (including greater than or equal to the optional user-defined minimum, if set) - see Section 20.2. If this is not the case, the server replies to the client with a ZCL 'default response' indicating an invalid value.

The change takes effect immediately. So, if the End Device is in normal poll mode when the 'long poll interval' is modified, the polling period is immediately re-timed to the new value.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster client resides
- *u8DestinationEndPointId*: Number of remote endpoint on which cluster server resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the server node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload*: Pointer to structure containing the payload for the command (see Section 20.9.3), including the desired long poll interval

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN

- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Client Functions

eCLD_PollControlSetShortPollIntervalSend

```
teZCL_Status eCLD_PollControlSetShortPollIntervalSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PollControl_SetShortPollIntervalPayload *psPayload);
```

Description This function can be used on a cluster client to send a 'Set Short Poll Interval' command to the cluster server. This command requests the 'short poll interval' for fast poll mode on the End Device to be set to the specified value.

On receiving the command, the 'short poll interval' attribute is only modified by the server if the specified value is within the valid range for the attribute (including greater than or equal to the optional user-defined minimum, if set) - see Section 20.2. If this is not the case, the server replies to the client with a ZCL 'default response' indicating an invalid value.

The change takes effect immediately. So, if the End Device is in fast poll mode when the 'short poll interval' is modified, the polling period is immediately re-timed to the new value.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster client resides
- *u8DestinationEndPointId*: Number of remote endpoint on which cluster server resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the server node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 20.9.4), including the desired short poll interval

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND

- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Client Functions

eCLD_PollControlFastPollStopSend

```
teZCL_Status eCLD_PollControlFastPollStopSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on a cluster client to send a 'Fast Poll Stop' command to the cluster server. This command is intended to abort a fast poll mode episode which has been started on the server as the result of a 'Check-in Response'. Therefore, the command allows fast poll mode to be exited before the mode's timeout is reached.

The cluster server only stops fast poll mode on the destination End Device if a matching 'Fast Poll Stop' command has been received for every request to start the current episode of fast poll mode. Therefore, if the current fast poll mode episode resulted from multiple start requests from multiple clients, the episode cannot be prematurely stopped (before the timeout is reached) unless a 'Fast Poll Stop' command is received from each of those clients.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoints on which the cluster client resides
- *u8DestinationEndPointId*: Number of remote endpoints on which the cluster server resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the server node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Client Functions

Parent topic: Functions

Parent topic:*Poll Control Cluster*

Return codes The Poll Control cluster functions use the ZCL return codes, listed in Section 7.2.

Parent topic:*Poll Control Cluster*

Enumerations

'Attribute ID' enumerations The following structure contains the enumerations used to identify the attributes of the Poll Control cluster.

```
typedef enum PACK
{
 E_CLD_POLL_CONTROL_ATTR_ID_CHECKIN_INTERVAL 0x0000,
 E_CLD_POLL_CONTROL_ATTR_ID_LONG_POLL_INTERVAL,
 E_CLD_POLL_CONTROL_ATTR_ID_SHORT_POLL_INTERVAL,
 E_CLD_POLL_CONTROL_ATTR_ID_FAST_POLL_TIMEOUT,
 E_CLD_POLL_CONTROL_ATTR_ID_CHECKIN_INTERVAL_MIN,
 E_CLD_POLL_CONTROL_ATTR_ID_LONG_POLL_INTERVAL_MIN,
 E_CLD_POLL_CONTROL_ATTR_ID_FAST_POLL_TIMEOUT_MAX;
}teCLD_PollControl_Cluster_AttrID;
```

Parent topic: Enumerations

'Command' Enumerations The following enumerations represent the commands that the Poll Control cluster generates.

```
typedef enum PACK
{
  E_CLD_POLL_CONTROL_CMD_CHECK_IN = 0x00,
  E_CLD_POLL_CONTROL_CMD_FAST_POLL_STOP,
  E_CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL,
  E_CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL,
} teCLD_PollControl_CommandID;
```

The above enumerations are used to indicate types of Poll Control cluster events and are described in Section 20.5.

Parent topic: Enumerations

Parent topic:*Poll Control Cluster*

Structures

tsCLD_PPCallBackMessage For a Poll Control event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_PollControlCallBackMessage structure:

```
typedef struct
{
 uint8 u8CommandId;
 union
 {
 tsCLD_PollControl_CheckinResponsePayload *psCheckinResponsePayload;
 #ifdef CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL
 tsCLD_PollControl_SetLongPollIntervalPayload
        *psSetLongPollIntervalPayload;
  #endif
  #ifdef CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL
  tsCLD_PollControl_SetShortPollIntervalPayload
        *psSetShortPollIntervalPayload;
  #endif
 } uMessage;
} tsCLD_PollControlCallBackMessage;
```

where:

- u``8CommandId indicates the type of Poll Control command that has been received, one of:
  - E_CLD_POLL_CONTROL_CMD_CHECK_IN
  - E_CLD_POLL_CONTROL_CMD_FAST_POLL_STOP
  - E_CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL
  - E_CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL
    - If they are required, the last two commands must be enabled in the compile-time options, as described in Section 20.10.
- uMessage is a union containing the command payload, as one of (depending on the value of u8CommandId):
  - psCheckinResponsePayload is a pointer to the payload of a 'Check-in Response' (see Section 20.9.2)
  - psSetLongPollIntervalPayload is a pointer to the payload of a 'Set Long Poll Interval' command (see Section 20.9.3)
  - psSetShortPollIntervalPayload is a pointer to the payload of a 'Set Short Poll Interval' command (see Section 20.9.4)

The command payload for each command type is indicated in Table 29 in Section 20.5.

Parent topic: Structures

tsCLD_PollControl_CheckinResponsePayload This structure contains the payload of a 'Check-in Response', which is sent from the client to the server in reply to a 'Check-in' command from the server.

```
typedef struct
{
  zbool    bStartFastPolling;
  zuint16   u16FastPollTimeout;
}tsCLD_PollControl_CheckinResponsePayload;
```

where:

- bStartFastPolling is a boolean indicating whether or not the End Device is required to enter fast poll mode:
  - TRUE: Enter fast poll mode

  - FALSE: Continue in normal poll mode

- u16FastPollTimeout is an optional fast poll mode timeout, in quarter-seconds, in the range 1 to 65535 - that is, the period of time for which the End Device should remain in fast poll mode (if this mode is requested through bStartFastPolling). Zero is a special value which indicates that the value of the 'fast poll timeout' attribute should be used instead (see Section 20.2). If a non-zero value is specified then this value over-rides the 'fast poll timeout' attribute (but does not over-write it).

Parent topic: Structures

tsCLD_PollControl_SetLongPollIntervalPayload This structure contains the payload of a 'Set Long Poll Interval' command, which is sent from the client to the server to request a new 'long poll interval' for use in normal poll mode.

```
typedef struct
{
 zuint32  u32NewLongPollInterval;
}tsCLD_PollControl_SetLongPollIntervalPayload;
```

where u32NewLongPollInterval is the required value of the 'long poll interval', in quarter-seconds, in the range 4 to 7208960. This value is used to over-write the corresponding cluster attribute if the specified value is within the valid range for the attribute (including greater than or equal to the optional user-defined minimum, if set).

To use the 'Set Long Poll Interval' command, it must be enabled in the compile-time options, as described in Section 20.10.

Parent topic: Structures

tsCLD_PollControl_SetShortPollIntervalPayload This structure contains the payload of a 'Set Short Poll Interval' command, which is sent from the client to the server to request a new 'short poll interval' for use in fast poll mode.

```
typedef struct
{
 zuint16  u16NewShortPollInterval;
}tsCLD_PollControl_SetShortPollIntervalPayload;
```

where u16NewShortPollInterval is the required value of the 'short poll interval', in quarter-seconds, in the range 1 to 65535. This value is used to over-write the corresponding cluster attribute if the specified value is within the valid range for the attribute (including greater than or equal to the optional user-defined minimum, if set).

To use the 'Set Short Poll Interval' command, it must be enabled in the compile-time options, as described in Section 20.10.

Parent topic: Structures

tsCLD_PollControlCustomDataStructure The Poll Control cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
#ifdef POLL_CONTROL_SERVER
    tsCLD_PollControlParameters          sControlParameters;
```

(continues on next page)

(continued from previous page)

```
#endif
    tsZCL_ReceiveEventAddress            sReceiveEventAddress;
    tsZCL_CallBackEvent                  sCustomCallBackEvent;
    tsCLD_PollControlCallBackMessage     sCallBackMessage;
} tsCLD_PollControlCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Parent topic:*Poll Control Cluster*

Compile-time Options This section describes the compile-time options that may be configured in the **zcl_options.h** file of an application that uses the Poll Control cluster.

To enable the Poll Control cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_POLL_CONTROL
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define POLL_CONTROL_SERVER
#define POLL_CONTROL_CLIENT
```

The following options can also be configured at compile-time in the **zcl_options.h** file.

Optional Server Attributes To enable and assign a value (t quarter-seconds) to the optional Check-in Interval Minimum (u32CheckinIntervalMin) attribute, add the line:

```
#define CLD_POLL_CONTROL_ATTR_CHECKIN_INTERVAL_MIN t
```

To enable and assign a value (t quarter-seconds) to the optional Long Poll Interval Minimum (u32LongPollIntervalMin) attribute, add the line:

```
#define CLD_POLL_CONTROL_ATTR_LONG_POLL_INTERVAL_MIN t
```

To enable and assign a value (t quarter-seconds) to the optional Fast Poll Timeout Maximum (u16FastPollTimeoutMax) attribute, add the line:

```
#define CLD_POLL_CONTROL_ATTR_FAST_POLL_TIMEOUT_MAX t
```

Note: For further information on the above optional server attributes, refer to Section 20.2.

Global Attributes Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_POLL_CONTROL_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Set Valid Range for 'Check-in Interval' To set the maximum possible 'check-in interval' (to t quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_CHECKIN_INTERVAL_MAX t
```

The default value is 7208960.

To set the minimum possible 'check-in interval' (to t quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_CHECKIN_INTERVAL_MIN t
```

The default value is 0.

This minimum value is only applied if the Check-in Interval Minimum attribute (u32CheckinIntervalMin) is not enabled.

Set Valid Range for 'Fast Poll Timeout' To set the maximum possible 'fast poll timeout' (to t quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_FAST_POLL_TIMEOUT_MAX t
```

The default value is 65535.

To set the minimum possible 'fast poll timeout' (to t quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_FAST_POLL_TIMEOUT_MIN t
```

The default value is 1.

This maximum value is only applied if the Fast Poll Timeout Maximum attribute (u16FastPollTimeoutMax) is not enabled.

Set Valid Range for 'Long Poll Interval' To set the maximum possible 'long poll interval' (to t quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_LONG_POLL_INTERVAL_MAX t
```

The default value is 7208960.

To set the minimum possible 'long poll interval' (to t quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_LONG_POLL_INTERVAL_MIN t
```

The default value is 4.

This minimum value is only applied if the Long Poll Interval Minimum attribute (u32LongPollIntervalMin) is not enabled.

Set Valid Range for 'Short Poll Interval' To set the maximum possible 'short poll interval' (to t quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_SHORT_POLL_INTERVAL_MAX t
```

The default value is 65535.

To set the minimum possible 'short poll interval' (to t quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_SHORT_POLL_INTERVAL_MIN t
```

The default value is 1.

Optional Commands To enable the optional 'Set Long Poll Interval' command, add the line:

```
#define CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL
```

To enable the optional 'Set Short Poll Interval' command, add the line:

```
#define CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL
```

Maximum Number of Clients To set the maximum number of clients for a server to n, add the line:

```
#define CLD_POLL_CONTROL_NUMBER_OF_MULTIPLE_CLIENTS n
```

This is the maximum number of clients from which the server can handle Check-in Responses. It should be equal to the capacity (number of entries) of the binding table created on the server device to accommodate bindings to client devices (where this size is set in a ZigBee network parameter using the ZPS Configuration Editor).

Disable APS Acknowledgments for Bound Transmissions To disable APS acknowledgments for bound transmissions from this cluster, add the line:

```
#define CLD_POLL_CONTROL_BOUND_TX_WITH_APS_ACK_DISABLED
```

Parent topic:*Poll Control Cluster*

Power Profile Cluster This chapter describes the Power Profile cluster which provides an interface between a home appliance (e.g. a washing machine) and the controller of an energy management system.

The Power Profile cluster has a Cluster ID of 0x001A.

Overview The Power Profile cluster allows an appliance, the cluster server, to provide its expected power usage data to a controller, the cluster client. This 'power profile' represents the predicted 'energy footprint' of the appliance, and may be used by the controller to schedule and control the operation of the appliance. It may be requested by the client or provided unsolicited by the server.

The cluster is enabled by defining CLD_PP in the **zcl_options.h** file. Further compile-time options for the Power Profile cluster are detailed in Section 21.11.

Note: The Power Profile cluster requires the Appliance Control cluster for the implementation of status notifications and power management commands. The Appliance Control cluster is described in *Chapter 45*.

Parent topic:*Power Profile Cluster*

Cluster structure and attributes The structure definition for the Power Profile cluster (server) is:

```
typedef struct
{
#ifdef PP_SERVER
  zuint8  u8TotalProfileNum;
  zbool  bMultipleScheduling;
  zbmap8  u8EnergyFormatting;
  zbool  bEnergyRemote;
  zbmap8  u8ScheduleMode;
#ifdef CLD_PP_ATTR_ATTRIBUTE_REPORTING_STATUS
  zuint8 u8AttributeReportingStatus;
#endif
#endif
  zuint16  u16ClusterRevision;
} tsCLD_PP;
```

where:

- u8TotalProfileNum is the number of power profiles supported by the device (must be between 1 and 254, inclusive)
- bMultipleScheduling is a boolean indicating whether the server side of the cluster supports the scheduling of multiple energy phases or just a single energy phase at a time (according to commands received from the client):
  - TRUE if multiple energy phase scheduling is possible
  - FALSE if only single energy phase scheduling is possible
- u8EnergyFormatting indicates the format of the Energy fields in the Power Profile Notification and Power Profile Response:
  - Bits 0-2: Number of digits to the right of the decimal point
  - Bits 3-6: Number of digits to the left of the decimal point
  - Bit 7: If set to '1', any leading zeros are removed
- bEnergyRemote is a boolean indicating whether the cluster server (appliance) is configured for remote control (of energy management):
  - TRUE if at least one power profile is enabled for remote control
  - FALSE if no power profile is enabled for remote control
    - This attribute is linked to the bPowerProfileRemoteControl field in the power profile record (see Section 21.10.13) - if the latter field is set to TRUE, the attribute is also automatically set to TRUE.
- u8ScheduleMode indicates the criterion (cheapest or greenest) that should be used by the cluster client (for example, energy management system) to schedule the power profiles:
  - 0x00 - criterion is left to the cluster server to choose
  - 0x01 - cheapest mode (minimize cost of energy usage)
  - 0x02 - greenest mode (maximize use of renewable energy sources)
  - 0x03 - compromise between cheapest and greenest
    - All other values are reserved.
- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.

- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Power Profile Cluster*

Attributes for default reporting The following attributes of the Power Profile cluster can be selected for default reporting:

```
bEnergyRemote
u8ScheduleMode
```

Attribute reporting (including default reporting) is described in *Appendix B*. Enabling reports for these attributes is described in Appendix B.3.6.

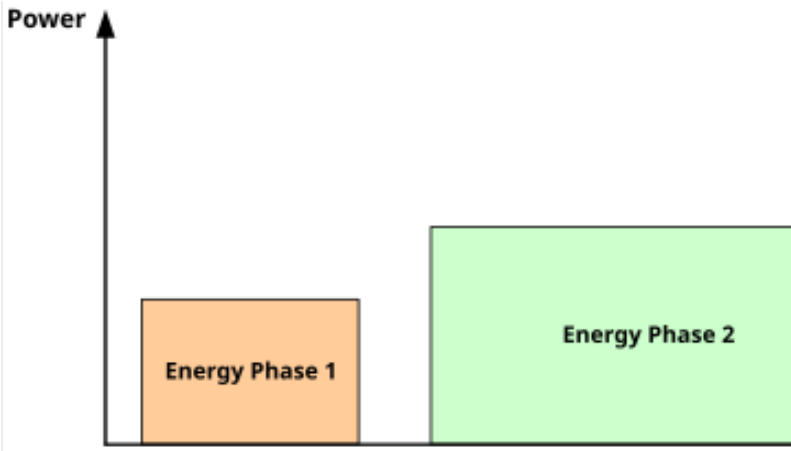
Parent topic:*Power Profile Cluster*

Power profiles An appliance can have one or more power profiles. An example of an appliance with multiple power profiles is a washing machine which has a number of programmes for different types of materials and loads.

Note: The number of power profiles on a device must be defined in the file **zcl_options.h** (see Section 21.11).

An individual power profile comprises a series of energy phases with different power demands. For example, these phases may correspond to the different cycles of a washing machine programme, such as wash, rinse, spin. Details of a power profile, including these energy phases, are held in an entry of the power profile table on the cluster server (appliance).

If the appliance is to be remotely controlled, the controller (cluster client) must 'learn' the details of the appliance's power profile so that it can control the scheduling of the energy phases. The schedule of a power profile is decided by the client, and includes energy phases and their relative start-times (the energy phases are not necessarily contiguous in time). A schedule is illustrated in the figure below. The client must communicate the schedule for a power profile to the server where the schedule is executed.



Schedule of Energy Phases of a Power Profile |

Parent topic:*Power Profile Cluster*

Power profile operations This section describes the main operations to be performed on the Power Profile cluster server (appliance) and client (controller).

Initialization The Power Profile cluster must be initialized on both the cluster server and client. This can be done using the function **eCLD_PPCreatePowerProfile()**, which creates an instance of the Power Profile cluster on a local endpoint.

If you are using a Zigbee device that includes the Power Profile cluster, the above function is automatically called by the initialization function for the device. The function **eCLD_PPCreatePowerProfile()** should be called explicitly when setting up a custom endpoint containing one or more selected clusters rather than the whole set of clusters supported by a standard Zigbee device.

Parent topic: Power profile operations

Adding and removing a power profile (server only) A Power Profile cluster server (appliance) supports one or more power profiles. Information on these power profiles is held on the server in a power profile table, where each table entry contains information on one supported power profile.

The application on the appliance can perform various operations on the power profile table, as described in the sub-sections below.

Adding a power profile entry The server application can introduce a new power profile by adding a corresponding entry to the power profile table using the function **eCLD_PPAddPowerProfileEntry()**. The new power profile table entry is specified in a tsCLD_PPEntry structure (see Section 21.10.2) supplied to this function. This structure includes the Power Profile ID - these identifiers should be numbered consecutively from 1 to 255.

The function **eCLD_PPAddPowerProfileEntry()** can also be used to replace (over-write) an existing power profile table entry, in which case the new entry should have the same Power Profile ID as the existing entry to be replaced.

Parent topic: Adding and removing a power profile (server only)

Removing a power profile entry The server application can remove a power profile from the device by calling the function **eCLD_PPRemovePowerProfileEntry()** to delete the corresponding entry of the local power profile table. The entry to be deleted is specified by means of the relevant Power Profile ID.

Parent topic: Adding and removing a power profile (server only)

Obtaining a Power Profile Entry The server application can obtain the details of a power profile supported by the server by reading the corresponding entry of the power profile table using the function **eCLD_PPGetPowerProfileEntry()**. The required entry is specified by means of the relevant Power Profile ID.

Parent topic: Adding and removing a power profile (server only)

Parent topic: Power profile operations

Communicating power profiles In order to control the power consumption of the appliance (by scheduling the energy phases of the power profile), the controller (cluster client) must 'learn' the power profiles supported by the appliance (server). This may be done through requests or notifications, as described in the sub-sections below.

Note: In order remotely control the appliance from a controller for energy management, the attribute bEnergyRemote of the Power Profile cluster on the server device must be set to TRUE (see Section 21.2).

Requesting a power profile (by client) A client application can request a power profile supported by the server by calling the **eCLD_PPPowerProfileReqSend()** function, which sends a Power Profile Request to the server. The function can be used to request a specific power profile (specified using its Power Profile ID) or all the power profiles supported by the server.

On receiving a response from the server, an E_CLD_PP_CMD_POWER_PROFILE_RSP event is generated on the client for each energy phase within the power profile. The reported information is contained in a tsCLD_PP_PowerProfilePayload structure (see Section 21.10.4). The application may store or discard this information, as required. By receiving the energy phase information in individual events, the application only needs to use as much memory as is required to store the relevant energy phase data.

Note: The client application may first use the function **eCLD_PPPowerProfileStateReqSend()** to request the identifiers of the power profiles that are currently supported on the server.

Parent topic: Communicating power profiles

Notification of a power profile (by server) The cluster server may send unsolicited notifications of the power profiles that it supports to the client. To do this, the server application must call the function **eCLD_PPPowerProfileNotificationSend()** which sends a Power Profile Notification containing the essential details of one supported power profile (such as the energy phases within the profile). This information is supplied to the function in a tsCLD_PP_PowerProfilePayload structure (see Section 21.10.4). If the server supports multiple power profiles, a separate notification must be sent for each profile.

On receiving the notification on the client, the event E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION is generated on the client for each energy phase within the power profile. The reported information is contained in a tsCLD_PP_PowerProfilePayload structure (see Section 21.10.4). The application may store or discard this information, as required. By receiving the energy phase information in individual events, the application only needs to use as much memory as is required to store the relevant energy phase data.

Parent topic: Communicating power profiles

Parent topic: Power profile operations

Communicating schedule information A power profile schedule comprises a sequence of energy phases and their relative start-times (the energy phases may have gaps between them):

- An energy phase is identified by its Energy Phase Identifier, in the range 1 to 255 (inclusive).
- The start-time of an energy phase is expressed as a delay, in minutes, from the end of the previous energy phase. For the first energy phase of a power profile schedule, this delay is measured from the time that the schedule was started.

Note: The normal duration of an energy phase, in minutes, is fixed and is specified in the energy phase information in the power profile.

Although a power profile on the cluster server may support multiple energy phases, the schedule for the power profile may possibly incorporate only a sub-set of these phases. The client (controller) selects the set of energy phases in a schedule and communicates this schedule to the server (appliance). This may be done through a request or notification, as described in Section 21.5.4.1 and Section 21.5.4.2 below.

Requesting a schedule (by server) The server application can request a schedule for a supported power profile from the client by calling the function **eCLD_PPEnergyPhasesScheduleReqSend()**, which sends an Energy Phases Schedule Request to the client.

The client can only return the requested schedule information if it stores this type of information for the power profile. If this is the case, an E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_REQ

event is generated on the client, with the bIsInfoAvailable field set to TRUE in the event structure tsCLD_PPCallBackMessage, and the client will send an Energy Phases Schedule Response back to the server. Otherwise, the client sends a ZCL default response with status NOT_FOUND.

On receiving an Energy Phases Schedule Response from the client, the event E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_RSP is generated on the server, containing the requested schedule information in a tsCLD_PP_EnergyPhasesSchedulePayload structure (see Section 21.10.6). One or more of the following outcomes result:

- If the attributebEnergyRemote is set to FALSE on the server (no remote control of the device), the server simply rejects the received schedule.
- If the received schedule information contains an u16MaxActivationDelay value of zero for an energy phase (see tsCLD_PP_EnergyPhaseInfo), this energy phase is rejected by the server although other valid energy phases are accepted. For each rejected energy phase, the server sends a ZCL default response with status NOT_AUTHORIZED to the client.
- If the received schedule information results in an update of the power profile schedule on the server, the server automatically sends an Energy Phases Schedule State Notification back to the client. On receiving this notification, an E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION event is generated on the client.

Note: Before requesting a power profile schedule, the server application may send the schedule's timing constraints to the client using the function **eCLD_PPPowerProfileScheduleConstraintsNotificationSend()**. The client application can alternatively request these schedule constraints from the server by calling **eCLD_PPPowerProfileScheduleConstraintsReqSend()**.

Parent topic: Communicating schedule information

Notification of a Schedule (by Client) The cluster client may send an unsolicited notification of a power profile schedule to the server. To do this, the client application must call the function **eCLD_PPEnergyPhasesScheduleNotificationSend()** which sends an Energy Phases Schedule Notification containing the schedule. This information is supplied to the function in a tsCLD_PP_EnergyPhasesSchedulePayload structure (see Section 21.10.6).

On receiving the notification on the server, the event E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_NOTIFICATI is generated, containing the sent power profile schedule. One or more of the following outcomes will result:

- If the attribute bEnergyRemote is set to FALSE on the server (no remote control of the device), the server will simply reject the received schedule.
- If the received schedule information contains an u16MaxActivationDelay value of zero for an energy phase (see Section 21.10.11), this energy phase is rejected by the server although other valid energy phases will be accepted. For each rejected energy phase, the server will send a ZCL default response with status NOT_AUTHORIZED to the client.
- If the received schedule information results in an update of the power profile schedule on the server, the server will automatically send an Energy Phases Schedule State Notification back to the client. On receiving this notification, an E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION event is generated on the client.

Parent topic: Communicating schedule information

Notification of Energy Phases in Power Profile Schedule (by Server) The server application can use the function **eCLD_PPEnergyPhasesScheduleStateNotificationSend()** to send an unsolicited Energy Phases Schedule State Notification to a cluster client, in order to inform the client of the energy phases that are in the schedule of a particular power profile.

Parent topic: Communicating schedule information

Requesting the Scheduled Energy Phases (by Client) The client application can use the function **eCLD_PPEnergyPhasesScheduleStateReqSend()** to send an Energy Phases Schedule State Request to the cluster server, in order to obtain the schedule of energy phases for a particular power profile on the server.

On receiving the response on the client, the event E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP is generated, containing the requested schedule information. The obtained schedule can be used to re-align the schedule information on the client with the information on the server - for example, after a reset of the client device.

Parent topic: Communicating schedule information

Parent topic: Power profile operations

Executing a Power Profile Schedule After receiving a power profile schedule from the client (as described in Section21.5.4), the server can start execution of the schedule. The instruction to start the schedule comes from the client in the form of an Energy Phases Schedule Notification. To issue this instruction, the client application must call the function **eCLD_PPEnergyPhasesScheduleNotificationSend()**. On receiving the notification, the server automatically starts the schedule.

The possible states of a power profile are fully detailed in Section 21.9.2 but, generally, it moves through the following principal states before, during and after execution:

- E_CLD_PP_STATE_PROGRAMMED: The power profile is defined in the local power profile table but a schedule has not been received from the client. Even without a schedule from the client, a schedule of energy phases that was defined when the power profile was introduced using the function **eCLD_PPAddPowerProfileEntry()** can be started from this state (see below).
- E_CLD_PP_STATE_WAITING_TO_START: The power profile remains in this state before the first energy phase starts and between energy phases (provided there is a gap between the end of one phase and the beginning of the next).
- E_CLD_PP_STATE_RUNNING: An energy phase is running.
- E_CLD_PP_STATE_ENDED: The final energy phase has completed.

Once a schedule has started, the server application must progress execution through the different states of the schedule by periodically calling the function **eCLD_PPSchedule()** once per second. This function moves the power profile to the next state, if it is due to start, and update the relevant state and timing parameters.

Note: The server application can also use the function **eCLD_PPSetPowerProfileState()** to 'manually' move execution of the schedule to a particular (valid) state, irrespective of whether the target state is scheduled. This function can be used by the server application to locally start a schedule from the 'programmed' state.

Whenever there is a change of state of a power profile, the cluster server automatically sends a Power Profile State Notification to the client. The server application can also send such a notification 'manually' by calling the function **eCLD_PPPowerProfileStateNotificationSend()**. The notification contains a power profile record that specifies the active power profile, the energy phase that is currently running (or due to run next) and the current state of the power profile. These notifications allow the controller to monitor the appliance. On receiving a notification on the client, an E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION event is generated, containing the sent power profile state information in a tsCLD_PP_PowerProfileStatePayload structure (see Section 21.10.5).

Parent topic: Power profile operations

Communicating Price Information The cost of implementing a power profile schedule on an appliance (cluster server) is determined/calculated by the controller (cluster client). The server can request price information from the client in a number of ways, as described below.

Note: Use of the Power Profile Price functions, referenced below, must be enabled in the compile-time options, as described in Section 21.11.

Requesting Cost of a Power Profile Schedule (by Server) The server application can use the function **eCLD_PPGetPowerProfilePriceSend()** to send a Get Power Profile Price Request to the client, in order to request the cost of executing the schedule of a particular power profile.

The client can only return the requested information if price-related information about the power profile is held on the client device. If this is the case, an E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE event is generated on the client, with the bIsInfoAvailable field set to TRUE in the event structure tsCLD_PPCallBackMessage and the client sends a Get Power Profile Price Response back to the server. Otherwise, the client sends a ZCL default response with status NOT_FOUND.

On receiving a Get Power Profile Price Response on the server, the event E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP is generated, containing the requested price information (if available).

Alternatively, the server application can use the function **eCLD_PPGetPowerProfilePriceExtendedSend()** to send a Get Power Profile Price Extended Request to a cluster client, in order to request specific cost information about a power profile supported by the server. The cost of executing a power profile can be requested with either scheduled energy phases or contiguous energy phases (no gaps between them). This request is handled by the client as described above for an ordinary Get Power Profile Price Request. However, the response results in an E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP event on the server, containing the requested price information (if available).

Parent topic: Communicating Price Information

Requesting Cost of Power Profile Schedules Over a Day (by Server) The server application can use the **eCLD_PPGetOverallSchedulePriceSend()** function to send a Get Overall Schedule Price Request to the client, in order to obtain the overall cost of all the power profiles that will be executed over the next 24 hours.

The client can only return the requested information if price-related information about the relevant power profiles is held on the client device. If this is the case, an E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE event is generated on the client, with the bIsInfoAvailable field set to TRUE in the event structure tsCLD_PPCallBackMessage. Otherwise, the client will generate a ZCL default response with status NOT_FOUND.

On receiving a Get Overall Schedule Price Response on the server, the event E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP is generated, containing the requested price information (if available).

Parent topic: Communicating Price Information

Parent topic: Power profile operations

Parent topic:*Power Profile Cluster*

Power Profile Events The Power Profile cluster has its own events that are handled through the callback mechanism described in *Chapter 3*. The cluster contains its own event handler. However, if a device uses this cluster then application-specific Power Profile event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function is then invoked when a Power Profile event occurs and needs the attention of the application.

For a Power Profile event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_PPCallBackMessage structure:

```
typedef struct
{
 uint8  u8CommandId;
#ifdef PP_CLIENT
  bool  bIsInfoAvailable;
#endif
  union
   {
   tsCLD_PP_PowerProfileReqPayload
                  *psPowerProfileReqPayload;
   tsCLD_PP_GetPowerProfilePriceExtendedPayload
       *psGetPowerProfilePriceExtendedPayload;
   } uReqMessage;
   union
   {
   tsCLD_PP_GetPowerProfilePriceRspPayload
                  *psGetPowerProfilePriceRspPayload;
   tsCLD_PP_GetOverallSchedulePriceRspPayload
      *psGetOverallSchedulePriceRspPayload;
    tsCLD_PP_EnergyPhasesSchedulePayload
                  *psEnergyPhasesSchedulePayload;
    tsCLD_PP_PowerProfileScheduleConstraintsPayload
      *psPowerProfileScheduleConstraintsPayload;
    tsCLD_PP_PowerProfilePayload
                  *psPowerProfilePayload;
       tsCLD_PP_PowerProfileStatePayload
                  *psPowerProfileStatePayload;
    }uRespMessage;
} tsCLD_PPCallBackMessage;
```

The above structure is fully described in Section 21.10.1.

When a Power Profile event occurs, one of the command types listed in *Table 30* and *Table 31* is specified through the u8CommandId field of the tsCLD_PPCallBackMessage structure. This command type determines which command payload is used from the unions uReqMessage (for request commands) and uRespMessage (for response and notification commands).

u8CommandId Enumeration	Description/Payload Type
E_CLD_PP_CMD_POWE	The server (appliance) receives a Power Profile Request. **tsCLD_PP_PowerProfileReqPayload**
E_CLD_PP_CMD_POWE	The server (appliance) receives a Power Profile State Request.
E_CLD_PP_CMD_GET_P	The server (appliance) receives a Get Power Profile Price Response, following a previously sent Get Power Profile Price Request. **tsCLD_PP_GetPowerProfilePriceRspPayload**
E_CLD_PP_CMD_GET_P	The server (appliance) receives a Get Power Profile Price Extended Response, following a previously sent Get Power Profile Price Extended Request.**tsCLD_PP_GetPowerProfilePriceRspPayload**

| |E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP|The server (appliance) receives a Get Overall Schedule Price Response, following a previously sent Get Overall Schedule Price Request.

tsCLD_PP_GetOverallSchedulePriceRspPayload

| |E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_NOTIFICATION|The server (appliance) re-

ceives an Energy Phases Schedule Notification.

tsCLD_PP_EnergyPhasesSchedulePayload

| |E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_RSP|The server (appliance) receives an Energy Phases Schedule Response, following a previously sent Energy Phases Schedule Request.

tsCLD_PP_EnergyPhasesSchedulePayload

| |E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP|The server (appliance) receives an Energy Phases Schedule State Response, following a previously sent Energy Phases Schedule State Request.

tsCLD_PP_EnergyPhasesSchedulePayload

| |E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_REQ|The server (appliance) receives a Get Power Profile Schedule Constraints Request.

tsCLD_PP_PowerProfileReqPayload

| |E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_REQ|The server (appliance) receives an Energy Phases Schedule State Request.

tsCLD_PP_PowerProfileReqPayload

|

u8CommandId Enumeration	Description/Payload Type
E_CLD_PP_CMD_POWER_PROFILE_NOTIFI	The client (controller) receives a Power Profile Notification.

tsCLD_PP_PowerProfilePayload

| |E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION|The client (controller) receives a Power Profile State Notification.

tsCLD_PP_PowerProfileStatePayload

| |E_CLD_PP_CMD_POWER_PROFILE_RSP|The client (controller) receives a Power Profile Response, following a previously sent Power Pro-file Request.

tsCLD_PP_PowerProfilePayload

| |E_CLD_PP_CMD_POWER_PROFILE_STATE_RSP|The client (controller) receives a Power Profile State Response, following a previously sent Power Profile State Request.

tsCLD_PP_PowerProfileStatePayload

| |E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE|The client (controller) receives a Get Power Profile Price Request.

tsCLD_PP_PowerProfileReqPayload

| |E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE|The client (controller) receives a Get Overall Schedule Price Request.| |E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_REQ|The client (controller) receives an Energy Phases Schedule Request .

tsCLD_PP_PowerProfileReqPayload

| |E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION|The client (controller) receives an Energy Phases Schedule State Notification.

tsCLD_PP_EnergyPhasesSchedulePayload

| |E_CLD_PP_CMD_SCHEDULE_CONSTRAINTS_NOTIFICATION|The client (controller) receives a Power Profile Schedule Constraints Notification.

tsCLD_PP_PowerProfileScheduleConstraintsPayload

| |E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_RSP|The client (controller) receives a Power Profile Schedule Constraints Response.

tsCLD_PP_PowerProfileScheduleConstraintsPayload

| |E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED|The client (controller) receives a Get Power Profile Price Extended Request.

tsCLD_PP_GetPowerProfilePriceExtendedPayload

|

Parent topic:*Power Profile Cluster*

Functions The Power Profile cluster functions are described in the following three subsections, according to the side(s) of the cluster on which they can be used:

- Server/client functions are described in Section 21.7.1
- Server functions are described in Section 21.7.2
- Client functions are described in Section 21.7.3

Server/Client Function The following Power Profile cluster function can be used on either a cluster server or cluster client:

- eCLD_PPCreatePowerProfile

eCLD_PPCreatePowerProfile

```
teZCL_Status eCLD_PPCreatePowerProfile(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits,
        tsCLD_PPCustomDataStructure *psCustomDataStructure);
```

Description This function creates an instance of the Power Profile cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard Zigbee device). This function creates a Power Profile cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to *Appendix D.*

Note: This function must not be called for an endpoint on which a standard Zigbee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in the *Zigbee Devices User Guide (JNUG3131).*

When used, this function must be the first Power Profile cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length is automatically adjusted by the compiler using the following declaration:

```
uint8 au8PPAttributeControlBits[(sizeof(asCLD_PPClusterAttrDefs)/
        sizeof(tsZCL_AttributeDefinition))];
int8 au8PPAttributeControlBits[(sizeof(asCLD_PPClusterAttrDefs)/
        sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Power Profile cluster. This parameter can refer to a pre-filled structure called sCLD_PP which is provided in the **PowerProfile.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_PP, which defines the attributes of the Power Profile cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 21.10.14)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Server/Client Function

Parent topic: Functions

Server Functions The following Power Profile cluster functions can be used on a cluster server only:

- eCLD_PPSchedule
- eCLD_PPSetPowerProfileState
- eCLD_PPAddPowerProfileEntry
- eCLD_PPRemovePowerProfileEntry
- eCLD_PPGetPowerProfileEntry
- eCLD_PPPowerProfileNotificationSend
- eCLD_PPEnergyPhaseScheduleStateNotificationSend
- eCLD_PPPowerProfileScheduleConstraintsNotificationSend

- eCLD_PPEnergyPhasesScheduleReqSend
- eCLD_PPPowerProfileStateNotificationSend
- eCLD_PPGetPowerProfilePriceSend
- eCLD_PPGetPowerProfilePriceExtendedSend
- eCLD_PPGetOverallSchedulePriceSend

eCLD_PPSchedule

```
teZCL_Status eCLD_PPSchedule(void);
```

Description This function can be used on a cluster server to update the state of the currently active power profile and the timings required for scheduling. When called, the function automatically makes any required changes according to the scheduled energy phases for the power profile. If no change is scheduled, the function only updates timing information. If a change is required, it also updates the power profile state and the ID of the energy phase currently being executed.

The function should be called once per second to progress the active power profile schedule and the application should provide a 1-second timer to prompt these function calls.

Parameters

- None

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Server Functions

eCLD_PPSetPowerProfileState

```
teZCL_CommandStatus eCLD_PPSetPowerProfileState(
    uint8 u8SourceEndPointId,
    uint8 u8PowerProfileId,
    teCLD_PP_PowerProfileState sPowerProfileState);
```

Description This function can be used on a cluster server to move the specified power profile to the specified target state. Enumerations for the possible states are provided, and are listed and described in Section 21.9.2.

The function performs the following checks:

- Checks whether the specified Power Profile ID exists (if not, the function returns with the status E_ZCL_CMDS_NOT_FOUND)
- Checks whether the specified target state is a valid state (if not, the function returns with the status E_ZCL_CMDS_INVALID_VALUE)

- Checks whether the power profile is currently able move to the target state (if not, the function returns with the status E_ZCL_CMDS_INVALID_FIELD)

Note: The power profile state can be changed by this function only if the move from the existing state to the target state is a valid change.

If all the checks are successful, the move is implemented (and the function returns with the status E_ZCL_CMD_SUCCESS).

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster resides
- *u8PowerProfileId* : Identifier of the power profile
- *sPowerProfileState*: Target state to which power profile is to be moved - enumerations are provided (see Section 21.9.2)

Returns

- E_ZCL_CMD_SUCCESS
- E_ZCL_CMDS_NOT_FOUND
- E_ZCL_CMDS_INVALID_VALUE
- E_ZCL_CMDS_INVALID_FIELD

Parent topic: Server Functions

eCLD_PPAddPowerProfileEntry

```
teZCL_Status eCLD_PPAddPowerProfileEntry(
    uint8 u8SourceEndPointId,
    tsCLD_PPEntry *psPowerProfileEntry);
```

Description This function can be used on a cluster server to introduce a new power profile by adding an entry to the local power profile table.

The function checks whether there is sufficient space in the table for the new power profile entry (if not, the function returns with the status E_ZCL_ERR_INSUFFICIENT_SPACE).

An existing power profile entry can be over-written with a new profile by specifying the same Power Profile ID (in the new entry structure).

The function also updates two of the cluster attributes (if needed), as follows.

- If a power profile is introduced which has multiple energy phases (as indicated by the u8NumOfScheduledEnergyPhases field of the tsCLD_PPEntry structure), the attribute bMultipleScheduling is set to TRUE (if not already TRUE)
- If a power profile is introduced which allows remote control for energy management (as indicated by the bPowerProfileRemoteControl field of the tsCLD_PPEntry structure), the attribute bEnergyRemote is set to TRUE (if not already TRUE)

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster resides:
- *psPowerProfileEntry* : Structure containing the power profile to add (see Section 21.10.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INSUFFICIENT_SPACE

Parent topic: Server Functions

eCLD_PPRemovePowerProfileEntry

```
teZCL_Status eCLD_PPRemovePowerProfileEntry(
    uint8 u8SourceEndPointId,
    uint8 u8PowerProfileId);
```

Description This function can be used on a cluster server to remove a power profile by deleting the relevant entry in the local power profile table.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster resides:
- *u8PowerProfileId* : Identifier of power profile to be removed

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Server Functions

eCLD_PPGetPowerProfileEntry

```
teZCL_Status eCLD_PPGetPowerProfileEntry(
    uint8 u8SourceEndPointId,
    uint8 u8PowerProfileId,
    tsCLD_PPEntry **ppsPowerProfileEntry);
```

Description This function can be used on a cluster server to obtain an entry from the local power profile table. The required entry is specified using the relevant Power Profile ID. The function obtains a pointer to the relevant entry, if it exists - a pointer must be provided to a location to receive the pointer to the entry.

If no entry with the specified Power Profile ID is found, the function returns E_ZCL_ERR_INVALID_VALUE.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster resides
- *u8PowerProfileId*: Identifier of power profile to be obtained
- *ppsPowerProfileEntry*: Pointer to a location to receive a pointer to the required power profile table entry (see Section 21.10.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Server Functions

eCLD_PPPowerProfileNotificationSend

```
teZCL_Status eCLD_PPPowerProfileNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_PowerProfilePayload *psPayload);
```

Description This function can be used on the cluster server to send a Power Profile Notification to a cluster client, in order to inform the client about one power profile supported by the server. The notification contains essential information about the power profile, including the energy phases supported by the profile (and certain details about them). If the server supports multiple power profiles, the function must be called for each profile separately.

On receiving the notification on the client, the event E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION is generated, containing the sent power profile information in a tsCLD_PP_PowerProfilePayload structure (see Section 21.10.4).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster server resides
- *u8DestinationEndPointId* : Number of remote endpoint on which cluster client resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the client node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload*: Pointer to structure containing the payload for the request (see Section 21.10.4), including essential information about the power profile

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Server Functions

eCLD_PPEnergyPhaseScheduleStateNotificationSend

```
teZCL_Status eCLD_PPEnergyPhasesScheduleStateNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_EnergyPhasesSchedulePayload
*psPayload);
```

Description This function can be used on the cluster server to send an Energy Phases Schedule State Notification to a cluster client, in order to inform the client of the energy phases that are in the schedule of a particular power profile. The function is used to send an unsolicited command.

On receiving the notification on the client, the event E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTI is generated, containing the sent power profile information in a tsCLD_PP_EnergyPhasesSchedulePayload structure (see Section 21.10.6).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster server resides
- *u8DestinationEndPointId*: Number of remote endpoint on which cluster client resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the client node
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- **psPayload*: Pointer to structure containing the payload for the request (see Section 21.10.6), including the identifier of the relevant power profile and the associated schedule of energy phases

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Server Functions

eCLD_PPPowerProfileScheduleConstraintsNotificationSend

```
teZCL_Status eCLD_PPPowerProfileScheduleConstraintsNotificationSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_PP_PowerProfileScheduleConstraintsPayload
```

(continues on next page)

(continued from previous page)

```
*psPayload);
```

Description This function can be used on the cluster server to send a Power Profile Schedule Constraints Notification to a cluster client, in order to inform the client of the schedule restrictions on a particular power profile. The constraints are specified in a tsCLD_PP_PowerProfileScheduleConstraintsPayload structure (see Section 21.10.7). They can subsequently be used by the client in calculating the schedule for the energy phases of the power profile. The function is used to send an unsolicited command.

On receiving the notification on the client, the event E_CLD_PP_CMD_SCHEDULE_CONSTRAINTS_NOTIFICATION is generated, containing the sent power profile constraint information in a tsCLD_PP_PowerProfileScheduleConstraintsPayload structure.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster server resides
- *u8DestinationEndPointId* : Number of remote endpoint on which cluster client resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the client node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload*: Pointer to structure containing the payload for the request (see Section 21.10.7), including the identifier of the relevant power profile and the associated schedule constraints

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Server Functions

eCLD_PPEnergyPhasesScheduleReqSend

```
teZCL_Status eCLD_PPEnergyPhasesScheduleReqSend(
uint8 u8SourceEndPointId,
uint8 u8DestinationEndPointId,
tsZCL_Address *psDestinationAddress,
uint8 *pu8TransactionSequenceNumber,
tsCLD_PP_PowerProfileReqPayload *psPayload);
```

Description This function can be used on the cluster server to send an Energy Phases Schedule Request to a cluster client, in order to obtain the schedule of energy phases for a particular power profile.

The function is non-blocking and will return immediately. On successfully receiving an Energy Phases Schedule Response from the client, an E_CLD_PP_CMD_ENERGY_PHASE_SCHEDULE_RSP event is generated on the server, containing the requested schedule information in a tsCLD_PP_EnergyPhasesSchedulePayload structure (see Section 21.10.6). For full details of handling an Energy Phases Schedule Request, refer to Section21.5.4.1.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster server resides:
- *u8DestinationEndPointId*: Number of remote endpoint on which cluster client resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the client node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload*: Pointer to structure containing the payload for the request (see Section 21.10.3), including the identifier of the relevant power profile

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Server Functions

eCLD_PPPowerProfileStateNotificationSend

```
teZCL_Status eCLD_PPEnergyPhasesScheduleStateNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_EnergyPhasesSchedulePayload
*psPayload);
```

Description This function can be used on the cluster server to send a Power Profile State Notification to a cluster client, in order to inform the client of the state of the power profile that is currently active on the server. The function is used to send an unsolicited command.

On receiving the notification on the client, the event E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION is generated, containing the sent power profile state information in a tsCLD_PP_PowerProfileStatePayload structure (see Section 21.10.5).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster server resides
- *u8DestinationEndPointId*: Number of remote endpoint on which cluster client resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the client node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload*: Pointer to structure containing the payload for the command (see Section 21.10.5), including the identifier and state of the relevant power profile

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Server Functions

eCLD_PPGetPowerProfilePriceSend

```
teZCL_Status eCLD_PPGetPowerProfilePriceSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_PowerProfileReqPayload *psPayload);
```

Description This function can be used on the cluster server to send a Get Power Profile Price Request to a cluster client, in order to request the cost of executing the schedule of a particular power profile. Use of this function must be enabled in the cluster compile-time options, as described in Section 21.11.

The function is non-blocking and will return immediately. On successfully receiving a Get Power Profile Price Response from the client, an E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP event is generated on the server, containing the requested price information in a tsCLD_PP_GetPowerProfilePriceRspPayload structure (see Section21.10.9). For full details of handling a Get Power Profile Price Request, refer to Section 21.5.6.1.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of local endpoint on which cluster server resides
- *u8DestinationEndPointId* Number of remote endpoint on which cluster client resides
- *psDestinationAddress* Pointer to a structure containing the destination address of the client node
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload* Pointer to structure containing the payload for the request (see Section 21.10.5), including the identifier of the relevant power profile

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Server Functions

eCLD_PPGetPowerProfilePriceExtendedSend

```
teZCL_Status eCLD_PPGetPowerProfilePriceExtendedSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_GetPowerProfilePriceExtendedPayload
*psPayload);
```

Description This function can be used on the cluster server to send a Get Power Profile Price Extended Request to a cluster client, in order to request specific cost information about a power profile supported by the server. The cost of executing a power profile can be requested with either scheduled energy phases or contiguous energy phases (no gaps between them). Use of this function must be enabled in the cluster compile-time options, as described in Section 21.11.

The function is non-blocking and will return immediately. On successfully receiving a Get Power Profile Price Extended Response from the client, an E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP event is generated on the server, containing the requested price information in a tsCLD_PP_GetPowerProfilePriceRspPayload structure (see Section21.10.9). For full details of handling a Get Power Profile Price Extended Request, refer to Section 21.5.6.1.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of local endpoint on which cluster server resides
- *u8DestinationEndPointId* Number of remote endpoint on which cluster client resides

- *psDestinationAddress* Pointer to a structure containing the destination address of the client node
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload* Pointer to structure containing the payload for the request (see Section 21.10.8), including the type of price information required

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Server Functions

eCLD_PPGetOverallSchedulePriceSend

```
teZCL_Status eCLD_PPGetOverallSchedulePriceSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on the cluster server to send a Get Overall Schedule Price Request to a cluster client, in order to obtain the overall cost of all the power profiles that are executed over the next 24 hours. Use of this function must be enabled in the cluster compile-time options, as described in Section 21.11.

The function is non-blocking and will return immediately. On successfully receiving a Get Overall Schedule Price Response from the client, an E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP event is generated on the server, containing the required price information in a tsCLD_PP_GetOverallSchedulePriceRspPayload structure (see Section21.10.10). For full details of handling a Get Overall Schedule Price Request, refer to Section 21.5.6.2.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster server resides
- *u8DestinationEndPointId*: Number of remote endpoint on which cluster client resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the client node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Server Functions

Parent topic: Functions

Client Functions The following Power Profile cluster functions can be used on a cluster client only:

- eCLD_PPPowerProfileRequestSend
- eCLD_PPEnergyPhasesScheduleNotificationSend
- eCLD_PPPowerProfileStateReqSend
- eCLD_PPEnergyPhasesScheduleStateReqSend
- eCLD_PPPowerProfileScheduleConstraintsReqSend

eCLD_PPPowerProfileRequestSend

```
teZCL_Status eCLD_PPPowerProfileRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_PowerProfileReqPayload *psPayload);
```

Description This function can be used on a cluster client to send a Power Profile Request to the cluster server, in order to obtain one or more power profiles from the server. The function can be used to request a specific power profile (specified using its identifier) or all the power profiles supported by the server (specified using an identifier of zero).

The function is non-blocking and will return immediately. On receiving the server's response, an E_CLD_PP_CMD_POWER_PROFILE_RSP event is generated on the client, containing a power profile in a tsCLD_PP_PowerProfilePayload structure (see Section 21.10.4).

When a particular Power Profile ID is specified but:

- The Power Profile ID is not in the valid range, the server will send a default response of INVALID_VALUE.
- The Power Profile ID is in the valid range but no data corresponding to this ID is available, the server will respond with a default response of NOT_FOUND.

When all power profiles on the server are requested, a response is received for each profile separately and, therefore, the above event is generated for each profile reported.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster client resides:
- *u8DestinationEndPointId* : Number of remote endpoint on which cluster server resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the server node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload*: Pointer to structure containing the payload for the request (see Section 21.10.3), including the identifier of the required power profile

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Client Functions

eCLD_PPEnergyPhasesScheduleNotificationSend

```
teZCL_Status eCLD_PPEnergyPhasesScheduleNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_EnergyPhasesSchedulePayload
*psPayload);
```

Description This function can be used on a cluster client to send an Energy Phases Schedule Notification to the cluster server, in order to start the schedule of energy phases of a power profile on the server. The function is used to send an unsolicited command and should only be called if the server allows itself to be remotely controlled. The command payload specifies the identifiers of the required energy phases and includes the relative start-times of the phases (see Section 21.10.12).

On receiving the notification on the server, the event E_CLD_PP_CMD_ENERGY_PHASE_SCHEDULE_NOTIFICATIO is generated, containing the sent energy phase schedule information in a tsCLD_PP_EnergyPhasesSchedulePayload structure (see Section 21.10.6). The subsequent handling of this notification is detailed in Section 21.5.4.2.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster client resides:
- *u8DestinationEndPointId* : Number of remote endpoint on which cluster server resides

- *psDestinationAddress*: Pointer to a structure containing the destination address of the server node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload*: Pointer to structure containing the payload for the command (see Section 21.10.6), including the scheduled energy phases and start-times

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Client Functions

eCLD_PPPowerProfileStateReqSend

```
teZCL_Status eCLD_PPPowerProfileStateReqSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on a cluster client to send a Power Profile State Request to the cluster server, in order to obtain the identifier(s) of the power profile(s) currently supported on the server.

The function is non-blocking and returns immediately. On receiving the server's response, an E_CLD_PP_CMD_POWER_PROFILE_STATE_RSP event is generated on the client, containing the required identifier(s). The response contains the power profile records of all the supported power profiles on the server in a tsCLD_PP_PowerProfileStatePayload structure (see Section 21.10.5).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster client resides:
- *u8DestinationEndPointId* : Number of remote endpoint on which cluster server resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the server node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Client Functions

eCLD_PPEnergyPhasesScheduleStateReqSend

```
teZCL_Status eCLD_PPEnergyPhasesScheduleStateReqSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_PowerProfileReqPayload *psPayload);
```

Description This function can be used on a cluster client to send an Energy Phases Schedule State Request to the cluster server, in order to obtain the schedule of energy phases for a particular power profile on the server. The obtained schedule can be used to re-align the schedule information on the client with the information on the server - for example, after a reset of the client device.

The function is non-blocking and returns immediately. On receiving the server's response, an E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP event is generated on the client, containing the requested schedule information in a tsCLD_PP_EnergyPhasesSchedulePayload structure (see Section21.10.6).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster client resides
- *u8DestinationEndPointId* : Number of remote endpoint on which cluster server resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the server node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload*: Pointer to structure containing the payload for the request (see Section 21.10.3), including the identifier of the relevant power profile

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL

- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Client Functions

eCLD_PPPowerProfileScheduleConstraintsReqSend

```
teZCL_Status eCLD_PPPowerProfileScheduleConstraintsReqSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_PowerProfileReqPayload *psPayload);
```

Description This function can be used on a cluster client to send a Power Profile Schedule Constraints Request to the cluster server, in order to obtain the schedule restrictions on a particular power profile on the server. The obtained constraints can subsequently be used in calculating a schedule of energy phases for the power profile (e.g. before calling **eCLD_PPEnergyPhasesScheduleNotificationSend()**).

The function is non-blocking and will return immediately. The server will send a response to this request and on receiving this response, an E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_RSP event is generated on the client, containing the requested schedule constraints in a tsCLD_PP_PowerProfileScheduleConstraintsPayload structure (see Section 21.10.7).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster client resides
- *u8DestinationEndPointId* : Number of remote endpoint on which cluster server resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the server node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload*: Pointer to structure containing the payload for the request (see Section 21.10.3), including the identifier of the relevant power profile

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Client Functions

Parent topic: Functions

Parent topic:*Power Profile Cluster*

Return codes The Power Profile cluster functions use the ZCL return codes, listed in Section 7.2.

Parent topic:*Power Profile Cluster*

Enumerations

'Attribute ID' Enumerations The following structure contains the enumerations used to identify the attributes of the Power Profile cluster.

```
typedef enum PACK
{
    E_CLD_PP_ATTR_ID_TOTAL_PROFILE_NUM = 0x0000,
    E_CLD_PP_ATTR_ID_MULTIPLE_SCHEDULE,
    E_CLD_PP_ATTR_ID_ENERGY_FORMATTING,
    E_CLD_PP_ATTR_ID_ENERGY_REMOTE,
    E_CLD_PP_ATTR_ID_SCHEDULE_MODE
} teCLD_PP_Cluster_AttrID;
```

Parent topic: Enumerations

'Power Profile State' Enumerations The following enumerations represent the possible states of a power profile.

```
typedef enum PACK
{
   E_CLD_PP_STATE_IDLE    = 0x00,
   E_CLD_PP_STATE_PROGRAMMED,
   E_CLD_PP_STATE_RUNNING    = 0x02,
   E_CLD_PP_STATE_PAUSED,
   E_CLD_PP_STATE_WAITING_TO_START,
   E_CLD_PP_STATE_WAITING_PAUSED,
   E_CLD_PP_STATE_ENDED,
} teCLD_PP_PowerProfileState;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_PP_ST	Not all parameters of the power profile have yet been defined
E_CLD_PP_ST	In the programmed state, as all the parameters of the power pro-file have been defined but there is no schedule or a schedule exists but has not been started
E_CLD_PP_ST	The power profile is active and an energy phase is running
E_CLD_PP_ST	The power profile is active but the current energy phase is paused
E_CLD_PP_ST	The power profile is between two energy phases - one has ended and the next one has not yet started. If the next energy phase is the first energy phase of the schedule, this state indicates that schedule has been started but the first energy has not yet started
E_CLD_PP_ST	The power profile has been paused while in the 'waiting to start' state (described above)
E_CLD_PP_ST	The power profile schedule has finished

Parent topic: Enumerations

'Server-Generated Command' Enumerations The following enumerations represent the commands that can be generated by the cluster server.

```
typedef enum PACK
{
 E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION = 0x00,
 E_CLD_PP_CMD_POWER_PROFILE_RSP,
 E_CLD_PP_CMD_POWER_PROFILE_STATE_RSP,
 E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE,
 E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION,
 E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE,
 E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_REQ,
 E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP,
 E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION,
 E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_NOTIFICATION,
 E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_RSP,
 E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED
 } teCLD_PP_ServerGeneratedCommandID;
```

The above enumerations are used to indicate types of Power Profile cluster events and are described in Section 21.6.

Parent topic: Enumerations

'Server-Received Command' Enumerations The following enumerations represent the commands that can be received by the cluster server (and are therefore generated on the cluster client).

```
typedef enum PACK
{
  E_CLD_PP_CMD_POWER_PROFILE_REQ = 0x00,
  E_CLD_PP_CMD_POWER_PROFILE_STATE_REQ,
  E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP,
  E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP,
  E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_NOTIFICATION,
  E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_RSP,
  E_CLD_PP_CMD_POWER_PROFILE_SCHEDULE_CONSTRAINTS_REQ,
  E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_REQ,
  E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP
 } teCLD_PP_ServerReceivedCommandID;
```

The above enumerations are used to indicate types of Power Profile cluster events and are described in Section 21.6.

Parent topic: Enumerations

Parent topic:*Power Profile Cluster*

Structures

tsCLD_PPCallBackMessage For a Power Profile event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_PPCallBackMessage structure:

```
typedef struct
{
```

(continues on next page)

(continued from previous page)

```
    uint8 u8CommandId;
#ifdef PP_CLIENT
    bool bIsInfoAvailable;
#endif
union
{
    tsCLD_PP_PowerProfileReqPayload *psPowerProfileReqPayload;
    tsCLD_PP_GetPowerProfilePriceExtendedPayload
        *psGetPowerProfilePriceExtendedPayload;
} uReqMessage;
union
{    tsCLD_PP_GetPowerProfilePriceRspPayload *psGetPowerProfilePriceRspPayload;
    tsCLD_PP_GetOverallSchedulePriceRspPayload
            *psGetOverallSchedulePriceRspPayload;
    tsCLD_PP_EnergyPhasesSchedulePayload *psEnergyPhasesSchedulePayload;
    tsCLD_PP_PowerProfileScheduleConstraintsPayload
            *psPowerProfileScheduleConstraintsPayload;
    tsCLD_PP_PowerProfilePayload *psPowerProfilePayload;
    tsCLD_PP_PowerProfileStatePayload *psPowerProfileStatePayload;
}uRespMessage;
} tsCLD_PPCallBackMessage;
```

where:

- u``8CommandId indicates the type of Power Profile command that has been received, one of:
  - E_CLD_PP_CMD_POWER_PROFILE_REQ
  - E_CLD_PP_CMD_ POWER_PROFILE_STATE_REQ
  - E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP
  - E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP
  - E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_NOTIFICATION
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_RSP
  - E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_REQ
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_REQ
  - E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION
  - E_CLD_PP_CMD_POWER_PROFILE_RSP
  - E_CLD_PP_CMD_POWER_PROFILE_STATE_RSP
  - E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION
  - E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE
  - E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_REQ
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION
  - E_CLD_PP_CMD_SCHEDULE_CONSTRAINTS_NOTIFICATION
  - E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_RSP
  - E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED

- bIsInfoAvailable is a client-only boolean field which indicates whether the appropriate type of information (to which the event relates) is held on the client:

  TRUE if the information type is held on the client, FALSE otherwise

- uReqMessage is a union containing the command payload for a request, as one of (depending on the value of u8CommandId):
  - psPowerProfileReqPayload is a pointer to the payload of a Power Profile Request, a Get Power Profile Schedule Constraints Request, an Energy Phases Schedule Request, an Energy Phases Schedule State Request or a Get Power Profile Price Request (see Section 21.10.5).
  - psGetPowerProfilePriceExtendedPayload is a pointer to the payload of a Get Power Profile Price Extended Request (see Section 21.10.8).
- uRespMessage is a union containing the command payload for a response or notification, as one of (depending on the value of u8CommandId):
  - psGetPowerProfilePriceRspPayload is a pointer to the payload of a Get Power Profile Price Response or a Get Power Profile Price Extended Response (see Section 21.10.9)
  - psGetOverallSchedulePriceRspPayloadis a pointer to the payload of a Get Overall Schedule Price Response (seeSection 21.10.10).
  - psEnergyPhasesSchedulePayload is a pointer to the payload of an Energy Phases Schedule Response, an Energy Phases Schedule State Response, an Energy Phases Schedule Notification or an Energy Phases Schedule State Notification (see Section 21.6).
  - psPowerProfileScheduleConstraintsPayload is a pointer to the payload of a Power Profile Schedule Constraints Response or a Power Profile Schedule Constraints Notification (see Section 21.10.10).
  - psPowerProfilePayload is a pointer to the payload of a Power Profile Response or a Power Profile Notification (see Section 21.10.4).
  - psPowerProfileStatePayload is a pointer to the payload of a Power Profile State Response or a Power Profile State Notification (see Section 21.10.5).

Note: The command payload for each command type is indicated in Table 30 and Table 31 in Section 21.6.

Parent topic: Structures

tsCLD_PPEntry This structure contains the data for a power profile table entry.

```
typedef struct
{
  zuint8  u8PowerProfileId;
  zuint8  u8NumOfTransferredEnergyPhases;
  zuint8  u8NumOfScheduledEnergyPhases;
  zuint8  u8ActiveEnergyPhaseId;
  tsCLD_PP_EnergyPhaseDelay
      asEnergyPhaseDelay[CLD_PP_NUM_OF_ENERGY_PHASES];
   tsCLD_PP_EnergyPhaseInfo
      asEnergyPhaseInfo[CLD_PP_NUM_OF_ENERGY_PHASES];
   zbool    bPowerProfileRemoteControl;
   zenum8   u8PowerProfileState;
   zuint16  u16StartAfter;
   zuint16  u16StopBefore;
} tsCLD_PPEntry;
```

where:

- u8PowerProfileId is the identifier of the power profile in the range 1 to 255 (0 is reserved)

- u8NumOfTransferredEnergyPhases is the number of energy phases supported within the power profile
- u8NumOfScheduledEnergyPhases is the number of energy phases actually scheduled within the power profile (must be less than or equal to the value of u8NumOfTransferredEnergyPhases)
- u8ActiveEnergyPhaseId is the identifier of the energy phase that is currently active or will be active next (if currently between energy phases)
- asEnergyPhaseDelay[] is an array containing timing information on the scheduled energy phases, where each array element corresponds to one energy phase of the schedule (see Section 21.10.12)
- asEnergyPhaseInfo[] is an array containing various information on the supported energy phases, where each array element corresponds to one energy phase of the profile (see Section 21.10.11)
- bPowerProfileRemoteControl is a boolean indicating whether the power profile can be remotely controlled: TRUE if it can be remotely controlled, FALSE otherwise (this property directly affects the attribute bEnergyRemote)
- u8PowerProfileState is a value indicating the current state of the power profile (enumerations are provided - see Section 21.9.2)
- u16StartAfter is the minimum time-delay, in minutes, to be implemented between an instruction to start the power profile schedule and actually starting the schedule
- u16StopBefore is the maximum time-delay, in minutes, to be implemented between an instruction to stop the power profile schedule and actually stopping the schedule

Parent topic: Structures

tsCLD_PP_PowerProfileReqPayload This structure contains the payload of various power profile requests.

```
typedef struct
{
 zuint8    u8PowerProfileId;
}tsCLD_PP_PowerProfileReqPayload;
```

where u8PowerProfileId is the identifier of the power profile of interest.

Parent topic: Structures

tsCLD_PP_PowerProfilePayload This structure contains the payload of a Power Profile Response or of a Power Profile Notification, which reports the details of a power profile.

```
typedef struct
{
   zuint8        u8TotalProfileNum;
   zuint8        u8PowerProfileId;
   zuint8        u8NumOfTransferredPhases;
   tsCLD_PP_EnergyPhaseInfo *psEnergyPhaseInfo;
}tsCLD_PP_PowerProfilePayload;
```

where:

- u8TotalProfileNum is the total number of power profiles supported on the originating device
- u8PowerProfileId is the identifier of the power profile being reported

- u8NumOfTransferredPhases is the number of energy phases supported within the power profile
- psEnergyPhaseInfo is a pointer to a structure or an array of structures (see Section 21.10.11) containing information on the supported energy phases, where each array element corresponds to one energy phase of the profile

Parent topic: Structures

tsCLD_PP_PowerProfileStatePayload This structure contains the payload of a Power Profile State Response or of a Power Profile State Notification.

```
typedef struct
{
  zuint8                    u8PowerProfileCount;
  tsCLD_PP_PowerProfileRecord *psPowerProfileRecord;
}tsCLD_PP_PowerProfileStatePayload;
```

where:

- u8PowerProfileCount is the number of power profiles in the payload
- psPowerProfileRecord is a pointer to one or more power profile records (see Section 21.10.13):
  - For a Power Profile State Notification, it is a pointer to the power profile record of the currently active power profile on the server
  - For a Power Profile State Response, it is a pointer to an array of power profile records for all the supported power profiles on the server

Parent topic: Structures

tsCLD_PP_EnergyPhasesSchedulePayload This structure contains the payload of an Energy Phases Schedule Response, of an Energy Phases Schedule State Response or of an Energy Phases Schedule State Notification.

```
typedef struct
{
  zuint8    u8PowerProfileId;
  zuint8    u8NumOfScheduledPhases;
  tsCLD_PP_EnergyPhaseDelay
                    *psEnergyPhaseDelay;
}tsCLD_PP_EnergyPhasesSchedulePayload;
```

where:

- u8PowerProfileId is the identifier of the power profile being reported
- u8NumOfScheduledPhases is the number of energy phases within the power profile schedule
- psEnergyPhaseDelay is a pointer to an array containing timing information on the scheduled energy phases, where each array element corresponds to one energy phase of the schedule (see Section 21.10.12)

Parent topic: Structures

tsCLD_PP_PowerProfileScheduleConstraintsPayload This structure contains the payload of a Power Profile Schedule Constraints Response or of a Power Profile Schedule Constraints Notification, which reports the schedule restrictions on a particular power profile.

```
typedef struct
{
  zuint8  u8PowerProfileId;
  zuint16  u16StartAfter;
  zuint16  u16StopBefore;
} tsCLD_PP_PowerProfileScheduleConstraintsPayload;
```

where:

- u8PowerProfileId is the identifier of the power profile being reported
- u16StartAfter is the minimum time-delay, in minutes, to be implemented between an instruction to start the power profile schedule and actually starting the schedule
- u16StopBefore is the maximum time-delay, in minutes, to be implemented between an instruction to stop the power profile schedule and actually stopping the schedule

Parent topic: Structures

tsCLD_PP_GetPowerProfilePriceExtendedPayload This structure contains the payload of a Get Power Profile Price Extended Request, which requests certain price information relating to a particular power profile.

```
typedef struct
{
   zbmap8   u8Options;
   zuint8   u8PowerProfileId;
   zuint16   u16PowerProfileStartTime;
}tsCLD_PP_GetPowerProfilePriceExtendedPayload;
```

where:

- u8Options is a bitmap indicating the type of request:
  - If bit 0 is set to '1' then the u16PowerProfileStartTime field is used, otherwise it is ignored
  - If bit 1 is set to '0' then an estimated price is required for contiguous energy phases (with no gaps between them); if bit 1 is set '1' then an estimated price is required for the energy phases as scheduled (with any scheduled gaps between them)
- u8PowerProfileId is the identifier of the power profile
- u16PowerProfileStartTime is an optional value (see u8Options above) which indicates the required start-time for execution of the power profile, in minutes, as measured from the current time

Parent topic: Structures

tsCLD_PP_GetPowerProfilePriceRspPayload This structure contains the payload of a Get Power Profile Price Response, which is returned in reply to a Get Power Profile Price Request and a Get Power Profile Price Extended Request.

```
typedef struct
{
    zuint8   u8PowerProfileId;
    zuint16   u16Currency;
    zuint32   u32Price;
    zuint8   u8PriceTrailingDigits;
}tsCLD_PP_GetPowerProfilePriceRspPayload;
```

where:

- u8PowerProfileId is the identifier of the power profile
- u16Currency is a value representing the currency in which the price is quoted
- u32Price is the price as an integer value (without a decimal point)
- u8PriceTrailingDigits specifies the position of the decimal point in the price u32Price, by indicating the number of digits after the decimal point

Parent topic: Structures

tsCLD_PP_GetOverallSchedulePriceRspPayload This structure contains the payload of a Energy Phases Schedule Response, which contains the overall cost of all the power profiles that will be executed over the next 24 hours.

```
typedef struct
{
   zuint16    u16Currency;
   zuint32    u32Price;
   zuint8    u8PriceTrailingDigits;
}tsCLD_PP_GetOverallSchedulePriceRspPayload;
```

where:

- u16Currency is a value representing the currency in which the price is quoted
- u32Price represents the price as an integer value (without a decimal point)
- u8PriceTrailingDigits specifies the position of the decimal point in the price u32Price, by indicating the number of digits after the decimal point

Parent topic: Structures

tsCLD_PP_EnergyPhaseInfo This structure contains various pieces of information about a specific energy phase of a power profile.

```
typedef struct
{
    zuint8   u8EnergyPhaseId;
    zuint8   u8MacroPhaseId;
    zuint16   u16ExpectedDuration;
    zuint16   u16PeakPower;
    zuint16   u16Energy;
    zuint16   u16MaxActivationDelay;
}tsCLD_PP_EnergyPhaseInfo;
```

where:

- u8EnergyPhaseId is the identifier of the energy phase
- u8MacroPhaseId is a value that may be used to obtain a name/label for the energy phase for display purposes - for example, it may be the index of an entry in a table of ASCII strings
- u16ExpectedDuration is the expected duration of the energy phase, in minutes
- u16PeakPower is the estimated peak power of the energy phase, in Watts
- u16Energy is the estimated total energy consumption, in Joules, during the energy phase (u16PeakPower x u16ExpectedDuration x 60)

- u16MaxActivationDelay is the maximum delay, in minutes, between the end of the previous energy phase and the start of this energy phase - special values are as follows: 0x0000 if no delay possible, 0xFFFF if first energy phase

Note: If u16MaxActivationDelay is non-zero, a delayed start-time for the energy phase can be set through the structure tsCLD_PP_EnergyPhaseDelay (see Section 21.10.12).

Parent topic: Structures

tsCLD_PP_EnergyPhaseDelay This structure contains the start-time for a particular energy phase of a power profile.

```
typedef struct
{
    zuint8   u8EnergyPhaseId;
    zuint16  u16ScheduleTime;
}tsCLD_PP_EnergyPhaseDelay;
```

where:

- u8EnergyPhaseId is the identifier of the energy phase
- u16ScheduleTime is the start-time of the energy phase expressed as a delay, in minutes, from the end of the previous energy phase (for the first energy phase of a power profile schedule, this delay is measured from the start of the schedule)

Note: A delayed start-time for the energy phase can only be set through this structure if the field u16MaxActivationDelay of the structure tsCLD_PP_EnergyPhaseInfo for this energy phase is non-zero (see Section 21.10.11).

Parent topic: Structures

tsCLD_PP_PowerProfiIeRecord This structure contains information about the current state of a power profile.

```
typedef struct
{
    zuint8   u8PowerProfileId;
    zuint8   u8EnergyPhaseId;
    zbool    bPowerProfileRemoteControl;
    zenum8   u8PowerProfileState;
} tsCLD_PP_PowerProfileRecord;
```

where:

- u8PowerProfileId is the identifier of the power profile
- u8EnergyPhaseId is the identifier of the currently running energy phase or, if currently between energy phases, the next energy phase to be run
- bPowerProfileRemoteControl is a boolean indicating whether the power profile can be remotely controlled (from a client device): TRUE if it can be remotely controlled, FALSE otherwise
- u8PowerProfileState is an enumeration indicating the current state of the power profile (see Section 21.9.2)

Parent topic: Structures

tsCLD_PPCustomDataStructure The Power Profile cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
#ifdef (CLD_PP) && (PP_SERVER)
   bool     bOverrideRunning;
   uint8    u8ActSchPhaseIndex;
   tsCLD_PPEntryasPowerProfileEntry[CLD_PP_NUM_OF_POWER_PROFILES];
#endif
  tsZCL_ReceiveEventAddress     sReceiveEventAddress;
  tsZCL_CallBackEvent       sCustomCallBackEvent;
  tsCLD_PPCallBackMessage     sCallBackMessage;
} tsCLD_PPCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Parent topic:*Power Profile Cluster*

Compile-time Options This section describes the compile-time options that may be configured in the **zcl_options.h** file of an application that uses the Power Profile cluster.

To enable the Power Profile cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_PP
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define PP_SERVER
#define PP_CLIENT
```

The following options can also be configured at compile-time in the **zcl_options.h** file.

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_PP_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_PP_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Enabling 'Get Power Profile Price' Command The optional 'Get Power Profile Price' command can be enabled by adding:

```
#define CLD_PP_CMD_GET_POWER_PROFILE_PRICE
```

Enabling 'Get Power Profile Price Extended' Command The optional 'Get Power Profile Price Extended' command can be enabled by adding:

```
#define CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED
```

Enable 'Get Overall Schedule Price' Command The optional 'Get Overall Schedule Price' command can be enabled by adding:

```
#define CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE
```

Number of Power Profiles The number of power profiles on the local device can be defined as n by adding:

```
#define CLD_PP_NUM_OF_PROFILES n
```

Maximum Number of Energy Phases The maximum number of energy phases in a power profile can be defined as n by adding:

```
#define CLD_PP_NUM_OF_ENERGY_PHASES n
```

By default, this value is 3.

Disabling APS Acknowledgments for Bound Transmissions APS acknowledgements for bound transmissions from this cluster can be disabled by adding:

```
#define CLD_PP_BOUND_TX_WITH_APS_ACK_DISABLED
```

Parent topic:*Power Profile Cluster*

Diagnostics Cluster This chapter describes the Diagnostics cluster.

The Diagnostics cluster has a Cluster ID of 0x0B05.

Overview The Diagnostics cluster allows the operation of the ZigBee PRO stack to be followed over time. It provides a tool for monitoring the performance of individual network nodes, including the routing of packets through these nodes.

Note: It is strongly recommended that Diagnostics cluster server attributes are stored in persistent memory to allow performance data to be preserved through a device reset or power interruption.

To use the functionality of this cluster, you must include the file **Diagnostics.h** in your application and enable the cluster by defining CLD_DIAGNOSTICS in the **zcl_options.h** file.

A Diagnostics cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Diagnostics cluster are fully detailed in Section 22.5.

The information that can potentially be stored in this cluster is organized into the following attribute sets:

- Hardware information
- Stack/Network information

Currently, only three attributes from the Stack/Network Information attribute set are supported (see Section 22.2).

This cluster has no associated events. However, reads and writes of the cluster attributes may give rise to ZCL events (the application is responsible for checking that a written value is within the valid range for the target attribute).

Parent topic:*Diagnostics Cluster*

Diagnostics Structure and Attributes

The structure definition for the Diagnostics cluster is:

```
typedef struct
{
#ifdef DIAGNOSTICS_SERVER
 /* Hardware Information attribute set */
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_NUMBER_OF_RESETS
  uint16 u16NumberOfResets;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_PERSISTENT_MEMORY_WRITES
  uint16 u16PersistentMemoryWrites;
 #endif
 /* Stack/Network Information attribute set */
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_MAC_RX_BCAST
  uint32 u32MacRxBcast;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_BCAST
  uint32 u32MacTxBcast;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_MAC_RX_UCAST
  uint32 u32MacRxUcast;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_UCAST
  uint32 u32MacTxUcast;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_UCAST_RETRY
  uint16 u16MacTxUcastRetry;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_UCAST_FAIL
  uint16 u16MacTxUcastFail;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_RX_BCAST
  uint16 u16ApsRxBcast;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_TX_BCAST
  uint16 u16ApsTxBcast;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_RX_UCAST
  uint16 u16ApsRxUcast;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_TX_UCAST_SUCCESS
  uint16 u16ApsTxUcastSuccess;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_TX_UCAST_RETRY
  uint16 u16ApsTxUcastRetry;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_TX_UCAST_FAIL
```

(continues on next page)

(continued from previous page)

```
 uint16 u16ApsTxUcastFail;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_ROUTE_DISC_INITIATED
 uint16 u16RouteDiscInitiated;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_NEIGHBOR_ADDED
 uint16 u16NeighborAdded;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_NEIGHBOR_REMOVED
 uint16 u16NeighborRemoved;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_NEIGHBOR_STALE
 uint16 u16NeighborStale;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_JOIN_INDICATION
 uint16 u16JoinIndication;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_CHILD_MOVED
 uint16 u16ChildMoved;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_NWK_FC_FAILURE
 uint16 u16NWKFCFailure;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_FC_FAILURE
 uint16 u16APSFCFailure;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_UNAUTHORIZED_KEY
 uint16 u16APSUnauthorizedKey;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_NWK_DECRYPT_FAILURE
 uint16 u16NWKDecryptFailure;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_DECRYPT_FAILURE
 uint16 u16APSDecryptFailure;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_PACKET_BUFFER_ALLOCATE_FAILURE
 uint16 u16PacketBufferAllocateFailure;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_RELAYED_UCAST
 uint16 u16RelayedUcast;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_PHY_TO_MAC_QUEUE_LIMIT_REACHED
 uint16 u16PhyToMACQueueLimitReached;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_PACKET_VALIDATE_DROP_COUNT
 uint16 u16PacketValidateDropCount;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_AVERAGE_MAC_RETRY_PER_APS_MESSAGE_SENT
 uint16 u16AverageMACRetryPerAPSMessageSent;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_LAST_MESSAGE_LQI
 uint8 u8LastMessageLQI;
#endif
#ifdef CLD_DIAGNOSTICS_ATTR_ID_LAST_MESSAGE_RSSI
 int8 i8LastMessageRSSI;
#endif
#endif
 zuint16 u16ClusterRevision;
} tsCLD_Diagnostics;
```

where:

'Hardware Information' Attribute Set The following two attributes can be maintained by the application using the Attribute Access functions detailed in Section 5.2.

- u16NumberOfResets is an optional attribute which acts as a counter of device resets/restarts (note that a factory reset clears this attribute) - thus, the attribute value must be incremented on each restart.
- u16PersistentMemoryWrites is an optional attribute which acts as a counter of the number of writes to persistent memory - thus, the attribute value must be incremented on each write.

'Stack/Network Information' Attribute Set The following attributes must be updated by the application by calling the function **eCLD_DiagnosticsUpdate()** (see Section 22.3) either periodically (at the highest rate possible) or on receiving an appropriate event from the stack.

```
u32MacRxBcast is reserved for future use
u32MacTxBcast is reserved for future use
u32MacRxUcast is reserved for future use
u32MacTxUcast is reserved for future use
u16MacTxUcastRetry is reserved for future use
u16MacTxUcastFail is reserved for future use
u16ApsRxBcast is reserved for future use
u16ApsTxBcast is reserved for future use
u16ApsRxUcast is reserved for future use
u16ApsTxUcastSuccess is reserved for future use
u16ApsTxUcastRetry is reserved for future use
u16ApsTxUcastFail is reserved for future use
u16RouteDiscInitiated is reserved for future use
u16NeighborAdded is reserved for future use
u16NeighborRemoved is reserved for future use
u16NeighborStale is reserved for future use
u16JoinIndication is reserved for future use
u16ChildMoved is reserved for future use
u16NWKFCFailure is reserved for future use
u16APSFCFailure is reserved for future use
u16APSUnauthorizedKey is reserved for future use
u16NWKDecryptFailure is reserved for future use
u16APSDecryptFailure is reserved for future use
u16PacketBufferAllocateFailure is reserved for future use
u16RelayedUcast is reserved for future use
u16PhyToMACQueueLimitReached is reserved for future use
u16PacketValidateDropCount is reserved for future use
```

- u16AverageMACRetryPerAPSMessageSent is an optional attribute which is used to maintain a record of the average number of IEEE802.15.4 MAC-level retries needed to send a message from the APS layer of the stack.
- u8LastMessageLQI is an optional attribute containing the LQI (Link Quality Indicator) value for the last message received, as a value in the range 0 to 255 where 0 indicates the worst link quality and 255 indicates the best link quality.
- i8LastMessageRSSI is an optional attribute containing the RSSI (Receive Signal Strength Indication) value of the last message received.

Note: If the value of u8LastMessageLQI or i8LastMessageRSSI is read remotely, the returned value will relate to the received message that contained the instruction to read the attribute.

Global Attributes

- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 cor-

responds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Diagnostics Cluster*

Functions The following Diagnostics cluster functions are provided:

- eCLD_DiagnosticsCreateDiagnostics
- eCLD_DiagnosticsUpdate

The cluster attributes can also all be accessed using the general attribute read/write functions, as described in Section 2.3.

eCLD_DiagnosticsCreateDiagnostics

```
teZCL_Status eCLD_DiagnosticsCreateDiagnostics(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Diagnostics cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates a Diagnostics cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Diagnostics cluster. This parameter can refer to a pre-filled structure called sCLD_Diagnostics which is provided in the **Diagnostics.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_Diagnostics which defines the attributes of Diagnostics cluster. The function initializes the attributes with default values.

- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_DiagnosticsUpdate

```
teZCL_Status eCLD_DiagnosticsUpdate(
    uint8 u8SourceEndPointId);
```

Description This function updates the (three) Stack/Network Information attributes (see Section22.2). It should be called periodically by the application (on the cluster server) at the highest rate possible or when an appropriate stack event occurs.

The attributes can otherwise be accessed (for example, read) using the Attribute Access functions detailed in Section 5.2.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint on which cluster server resides

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND

Parent topic: Functions

Parent topic:*Diagnostics Cluster*

Enumerations

teCLD_Diagnostics_AttributeId The following structure contains the enumerations used to identify the attributes of the Diagnostics cluster.

```
typedef enum
{
 /* Hardware Information attribute IDs */
    E_CLD_DIAGNOSTICS_ATTR_ID_NUMBER_OF_RESETS = 0x0000,
    E_CLD_DIAGNOSTICS_ATTR_ID_PERSISTENT_MEMORY_WRITES,
 /* Stack/Network Information attribute IDs */
```

(continues on next page)

(continued from previous page)

```
    E_CLD_DIAGNOSTICS_ATTR_ID_MAC_RX_BCAST = 0x0100,
    E_CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_BCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_MAC_RX_UCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_UCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_UCAST_RETRY,
    E_CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_UCAST_FAIL,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_RX_BCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_TX_BCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_RX_UCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_TX_UCAST_SUCCESS,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_TX_UCAST_RETRY,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_TX_UCAST_FAIL,
    E_CLD_DIAGNOSTICS_ATTR_ID_ROUTE_DISC_INITIATED,
    E_CLD_DIAGNOSTICS_ATTR_ID_NEIGHBOR_ADDED,
    E_CLD_DIAGNOSTICS_ATTR_ID_NEIGHBOR_REMOVED,
    E_CLD_DIAGNOSTICS_ATTR_ID_NEIGHBOR_STALE,
    E_CLD_DIAGNOSTICS_ATTR_ID_JOIN_INDICATION,
    E_CLD_DIAGNOSTICS_ATTR_ID_CHILD_MOVED,
    E_CLD_DIAGNOSTICS_ATTR_ID_NWK_FC_FAILURE,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_FC_FAILURE,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_UNAUTHORIZED_KEY,
    E_CLD_DIAGNOSTICS_ATTR_ID_NWK_DECRYPT_FAILURE,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_DECRYPT_FAILURE,
    E_CLD_DIAGNOSTICS_ATTR_ID_PACKET_BUFFER_ALLOCATE_FAILURE,
    E_CLD_DIAGNOSTICS_ATTR_ID_RELAYED_UCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_PHY_TO_MAC_QUEUE_LIMIT_REACHED,
    E_CLD_DIAGNOSTICS_ATTR_ID_PACKET_VALIDATE_DROP_COUNT,
    E_CLD_DIAGNOSTICS_ATTR_ID_AVERAGE_MAC_RETRY_PER_APS_MESSAGE_SENT,
    E_CLD_DIAGNOSTICS_ATTR_ID_LAST_MESSAGE_LQI,
    E_CLD_DIAGNOSTICS_ATTR_ID_LAST_MESSAGE_RSSI
} teCLD_Diagnostics_AttributeId;
```

Parent topic: Enumerations

Parent topic:*Diagnostics Cluster*

Compile-time Options To enable the Diagnostics cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_DIAGNOSTICS
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one of the following to the same file:

```
#define DIAGNOSTICS_CLIENT
#define DIAGNOSTICS_SERVER
```

Optional Attributes Add this line to enable the optional Average MAC Retry Per APS Message Sent attribute:

```
#define CLD_DIAGNOSTICS_ATTR_ID_AVERAGE_MAC_RETRY_PER_APS_MESSAGE_SENT
```

Add this line to enable the optional Last Message LQI attribute:

```
#define CLD_DIAGNOSTICS_ATTR_ID_LAST_MESSAGE_LQI
```

Add this line to enable the optional Last Message RSSI attribute:

```
#define CLD_DIAGNOSTICS_ATTR_ID_LAST_MESSAGE_RSSI
```

Global Attributes Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_DIAGNOSTICS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Diagnostics Cluster*

Part IV: Measurement and Sensing Clusters This part comprises eight chapters:

- *Chapter 23* details the **Illuminance Measurement** cluster
- *Chapter 24* details the **Illuminance Level Sensing** cluster
- *Chapter 25* details the **Temperature Measurement** cluster
- *Chapter 26* details the **Pressure Measurement** cluster
- *Chapter 27* details the **Flow Measurement** cluster
- *Chapter 28* details the **Relative Humidity Measurement** cluster
- *Chapter 29* details the **Occupancy Sensing** cluster
- *Chapter 30* details the **Electrical Measurement** cluster

Illuminance Measurement Cluster This chapter describes the Illuminance Measurement cluster which is defined in the ZCL and provides an interface to a light sensor that is able to make illuminance measurements.

The Illuminance Measurement cluster has a Cluster ID of 0x0400.

Overview The Illuminance Measurement cluster provides an interface to an illuminance measuring device, allowing the configuration of measuring and the reporting of measurements.

To use the functionality of this cluster, you must include the file **IlluminanceMeasurement.h** in your application and enable the cluster by defining CLD_ILLUMINANCE_MEASUREMENT in the **zcl_options.h** file.

An Illuminance Measurement cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Illuminance Measurement cluster are fully detailed in Section 23.6.

Parent topic:*Illuminance Measurement Cluster*

Illuminance Measurement Structure and Attributes The structure definition for the Illuminance Measurement cluster is:

```
typedef struct
{
#ifdef ILLUMINANCE_MEASUREMENT_SERVER
   zuint16     u16MeasuredValue;
   zuint16     u16MinMeasuredValue;
   zuint16     u16MaxMeasuredValue;
#ifdef CLD_ILLMEAS_ATTR_TOLERANCE
  zuint16    u16Tolerance;
#endif
#ifdef CLD_ILLMEAS_ATTR_LIGHT_SENSOR_TYPE
   zenum8    eLightSensorType;
#endif
#ifdef CLD_ILLMEAS_ATTR_ATTRIBUTE_REPORTING_STATUS
   zenum8   u8AttributeReportingStatus;
#endif
#endif
  zuint16   u16ClusterRevision;
} tsCLD_IlluminanceMeasurement;
```

where:

- u16MeasuredValue is a mandatory attribute representing the measured illuminance in logarithmic form, calculated as *(10000 x log10Illuminance) + 1*, where the illuminance is measured in Lux (lx). The possible illumination values are in the range 1 lx to 3.576 x 106 lx, corresponding to attribute values of 1 to 0xFFFE. The following attribute values have special meaning:
  - 0x0000: Illuminance is too low to be measured.
  - 0xFFFF: Illuminance measurement is invalid.
    - The valid range of values of u16MeasuredValue can be restricted using the attributes u16MinMeasuredValue and u16MaxMeasuredValue below - in this case, the attribute can take any value in the range u16MinMeasuredValue to u16MaxMeasuredValue.
- u16MinMeasuredValue is a mandatory attribute representing a lower limit on the value of the attribute u16MeasuredValue. The value must be less than that of u16MaxMeasuredValue. The value 0xFFFF indicates that the attribute is unused.
- u16MaxMeasuredValue is a mandatory attribute representing an upper limit on the value of the attribute u16MeasuredValue. The value must be greater than that of u16MinMeasuredValue. The value 0xFFFF indicates that the attribute is unused.
- u16Tolerance is an optional attribute which indicates the magnitude of the maximum possible error in the value of the attribute u16MeasuredValue. The true value is in the range (u16MeasuredValue – u16Tolerance) to (u16MeasuredValue + u16Tolerance) .
- eLightSensorType is an optional attribute that indicates the type of light sensor to which the cluster is interfaced:
  - 0x00: Photodiode
  - 0x01: CMOS
  - 0x02–0x3F: Reserved
  - 0x40–0xFE: Reserved for manufacturer-specific light sensor types
  - 0xFF: Unknown
- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates

whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.

- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Illuminance Measurement Cluster*

Attributes for Default Reporting The following attributes of the Illuminance Measurement cluster can be selected for default reporting:

```
u16MeasuredValue
u16Tolerance
```

Attribute reporting (including default reporting) is described in *Appendix B*. Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic:*Illuminance Measurement Cluster*

Functions The following Illuminance Measurement cluster function is provided in the NXP implementation of the ZCL:

- eCLD_IlluminanceMeasurementCreateIlluminanceMeasurement

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

eCLD_IlluminanceMeasurementCreateIlluminanceMeasurement

```
teZCL_Status eCLD_IlluminanceMeasurementCreateIlluminanceMeasurement(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Illuminance Measurement cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates an Illuminance Measurement cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Illuminance Measurement cluster. The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* : Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Illuminance Measurement cluster. This parameter can refer to a pre-filled structure called sCLD_IlluminanceMeasurement which is provided in the **IlluminanceMeasurement.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_IlluminanceMeasurement, which defines the attributes of Illuminance Measurement cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic:*Illuminance Measurement Cluster*

Enumerations

teCLD_IM_ClusterID The following structure contains the enumerations used to identify the attributes of the Illuminance Measurement cluster.

```
typedef enum
{
  E_CLD_ILLMEAS_ATTR_ID_MEASURED_VALUE = 0x0000, /* Mandatory */
  E_CLD_ILLMEAS_ATTR_ID_MIN_MEASURED_VALUE, /* Mandatory */
  E_CLD_ILLMEAS_ATTR_ID_MAX_MEASURED_VALUE, /* Mandatory */
  E_CLD_ILLMEAS_ATTR_ID_TOLERANCE,
  E_CLD_ILLMEAS_ATTR_ID_LIGHT_SENSOR_TYPE
} teCLD_IM_ClusterID;
```

Parent topic: Enumerations

Parent topic:*Illuminance Measurement Cluster*

Compile-time options To enable the Illuminance Measurement cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_ILLUMINANCE_MEASUREMENT
```

In addition, to include the software for a cluster client or server, it is necessary to add one of the following to the same file:

```
#define ILLUMINANCE_MEASUREMENT_CLIENT
#define ILLUMINANCE_MEASUREMENT_SERVER
```

Optional Attributes Add this line to enable the optional Tolerance attribute:

```
#define CLD_ILLMEAS_ATTR_TOLERANCE
```

Add this line to enable the optional Light Sensor Type attribute:

```
#define CLD_ILLMEAS_ATTR_LIGHT_SENSOR_TYPE
```

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_ILLMEAS_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_ILLMEAS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Illuminance Measurement Cluster*

Illuminance Level Sensing Cluster This chapter describes the Illuminance Level Sensing cluster which is defined in the ZCL and provides an interface to light-level sensing functionality.

The Illuminance Level Sensing cluster has a Cluster ID of 0x0401.

- **Overview**
- **Cluster structure and attributes**
- **Attributes for Default Reporting**
- **Functions**
- **Enumerations**
- **Compile-time Options**

Temperature Measurement Cluster This chapter describes the Temperature Measurement cluster which is concerned with configuring and reporting temperature measurement.

The Temperature Measurement cluster has a Cluster ID of 0x0402.

Overview The Temperature Measurement cluster provides an interface to a temperature measuring device, allowing the configuration of measuring and the reporting of measurements.

To use the functionality of this cluster, you must include the file **TemperatureMeasurement.h** in your application and enable the cluster by defining CLD_TEMPERATURE_MEASUREMENT in the **zcl_options.h** file.

A Temperature Measurement cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Temperature Measurement cluster are fully detailed in Section 25.6.

Parent topic:*Temperature Measurement Cluster*

Temperature Measurement Structure and Attributes The structure definition for the Temperature Measurement cluster (server) is:

```
typedef struct
{
#ifdef TEMPERATURE_MEASUREMENT_SERVER
  zint16     i16MeasuredValue;
  zint16     i16MinMeasuredValue;
  zint16     i16MaxMeasuredValue;
#ifdef CLD_TEMPMEAS_ATTR_TOLERANCE
  zuint16  u16Tolerance;
#endif
#ifdef CLD_TEMPMEAS_ATTR_ATTRIBUTE_REPORTING_STATUS
  zenum8     u8AttributeReportingStatus;
#endif
#endif
 zuint16 u16ClusterRevision;
} tsCLD_TemperatureMeasurement;
```

where:

- i16MeasuredValue is a mandatory attribute representing the measured temperature in degrees Celsius, as follows:
  - i16MeasuredValue = 100 x temperature in degrees Celsius
  - The possible values are used as follows:
    - 0x0000 to 0x7FFF represent positive temperatures from 0°C to 327.67°C.
    - 0x8000 indicates that the temperature measurement is invalid.
    - 0x8001 to 0x954C are unused values.
    - 0x954D to 0xFFFF represent negative temperatures from -273.15°C to -1°C (in two's complement form).
  - This attribute is updated continuously as measurements are made.
- i16MinMeasuredValue is a mandatory attribute specifying the value of the attribute i16MeasuredValue which corresponds to the minimum possible temperature that can be measured. Its value must be less than that of the attribute i16MaxMeasuredValue (below). The special value 0x8000 indicates that the minimum is not known.
- i16MaxMeasuredValue is a mandatory attribute specifying the value of the attribute i16MeasuredValue which corresponds to the maximum possible temperature that can be measured. Its value must be greater than that of the attribute i16MinMeasuredValue (above). The special value 0x8000 indicates that the maximum is not known.

- u16Tolerance is an optional attribute which indicates the magnitude of the maximum possible error in the value of the attribute u16MeasuredValue. The true value is in the range (u16MeasuredValue – u16Tolerance) to (u16MeasuredValue + u16Tolerance).
- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Temperature Measurement Cluster*

Attributes for Default Reporting The following attributes of the Temperature Measurement cluster can be selected for default reporting:

```
i16MeasuredValue
u16Tolerance
```

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic:*Temperature Measurement Cluster*

Functions The following Temperature Measurement cluster function is provided in the NXP implementation of the ZCL:eCLD_TemperatureMeasurementCreateTemperatureMeasurement

eCLD_TemperatureMeasurementCreateTemperatureMeasurement

```
teZCL_Status  eCLD_TemperatureMeasurementCreateTemperatureMeasurement(
  tsZCL_ClusterInstance *psClusterInstance,
  bool_t bIsServer,
  tsZCL_ClusterDefinition *psClusterDefinition,
  void *pvEndPointSharedStructPtr,
  uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Temperature Measurement cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates a Temperature Measurement cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length is automatically adjusted by the compiler using the following declaration:

```
uint8 au8TemperatureMeasurementAttributeControlBits
        [(sizeof(asCLD_TemperatureMeasurementClusterAttributeDefinitions)
    /     sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Temperature Measurement cluster. This parameter can refer to a pre-filled structure called sCLD_TemperatureMeasurement which is provided in the **TemperatureMeasurement.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_TemperatureMeasurementwhich defines the attributes of Temperature Measurement cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic:*Temperature Measurement Cluster*

Enumerations

teCLD_TemperatureMeasurement_AttributeID The following structure contains the enumerations used to identify the attributes of the Temperature Measurement cluster.

```
typedef enum
{
 E_CLD_TEMPMEAS_ATTR_ID_MEASURED_VALUE = 0x0000, /* Mandatory */
 E_CLD_TEMPMEAS_ATTR_ID_MIN_MEASURED_VALUE,      /* Mandatory */
 E_CLD_TEMPMEAS_ATTR_ID_MAX_MEASURED_VALUE,      /* Mandatory */
 E_CLD_TEMPMEAS_ATTR_ID_TOLERANCE,
} teCLD_TemperatureMeasurement_AttributeID;
```

Parent topic: Enumerations

Parent topic:*Temperature Measurement Cluster*

Compile-time Options To enable the Temperature Measurement cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_TEMPERATURE_MEASUREMENT
```

In addition, to include the software for a cluster client or server, it is necessary to add one of the following to the same file:

```
#define TEMPERATURE_MEASUREMENT_CLIENT
#define TEMPERATURE_MEASUREMENT_SERVER
```

Optional Attribute Add this line to enable the optional Tolerance attribute:

```
#define CLD_TEMPMEAS_ATTR_TOLERANCE
```

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_TEMPMEAS_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_TEMPMEAS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Temperature Measurement Cluster*

Pressure Measurement Cluster This chapter outlines the Pressure Measurement cluster, which provides an interface for configuring and obtaining pressure measurements.

The Pressure Measurement cluster has a Cluster ID of 0x0403.

Overview The Pressure Measurement cluster provides an interface for configuring and querying devices that perform pressure measurements.

- The server is located on the device which makes the pressure measurements
- The client is located on another device and queries the server for measurements

The cluster is enabled by defining CLD_PRESSURE_MEASUREMENT in the **zcl_options.h** file. Further compile-time options for the Pressure Measurement cluster are detailed in Section 26.9.

The information that can potentially be stored in this cluster is organized into the following attribute sets:

- Pressure Measurement Information
- Extended Pressure Measurement Information
- Global

Note that not all of the above attribute sets are currently implemented in the NXP software and not all attributes within a supported attribute set are implemented (see Section 26.2 for the supported attribute sets and attributes).

Parent topic:*Pressure Measurement Cluster*

Cluster structure and attributes The structure definition for the Pressure Measurement cluster (server) is:

```
typedef struct
{
#ifdef PRESSURE_MEASUREMENT_SERVER
  zint16 i16MeasuredValue;
  zint16 i16MinMeasuredValue;
  zint16 i16MaxMeasuredValue;
#ifdef CLD_PRESSUREMEAS_ATTR_TOLERANCE
  zuint16 u16Tolerance;
#endif
#ifdef CLD_PRESSUREMEAS_ATTR_ATTRIBUTE_REPORTING_STATUS
  zenum8 u8AttributeReportingStatus;
#endif
#endif
    zuint16 u16ClusterRevision;
} tsCLD_PressureMeasurement;
```

where:

'Pressure Measurement Information' Attribute Set

- i16MeasuredValue is a mandatory attribute corresponding to 10 times the measured pressure, in units of kPa, in two's complement form. The range of possible values is 0x8001 (representing -3276.7 kPa) through 0x0000 (0 kPa) to 0x7FFF (representing +3276.7 kPa). The value 0x8000 is used to indicate that the measurement was invalid. In practice, the stored value is limited within the range i16MinMeasuredValue to i16MaxMeasuredValue (see below).
- i16MinMeasuredValue is a mandatory attribute representing a lower limit on the value that can be stored in i16MeasuredValue. It is a two's complement value in the range 0x8001 to 0x7FFE, and it must be less than the value of the attribute i16MaxMeasuredValue.
- i16MaxMeasuredValue is a mandatory attribute representing an upper limit on the value that can be stored in i16MeasuredValue. It is a two's complement value in the range 0x8002 to 0x7FFF, and it must be greater than the value of the attribute i16MinMeasuredValue.

Global Attributes

- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Pressure Measurement Cluster*

Initialization and Operation The Pressure Measurement cluster must be initialized on both the cluster server and client. This can be done using the function **eCLD_PressureMeasurementCreatePressureMeasurement()**, which creates an instance of the Pressure Measurement cluster on a local endpoint.

Once the cluster has been initialized, the application on the server should maintain the cluster attributes (see Section 26.2) with the pressure measurements made by the local device. The application on a client can remotely read these measured values using the ZCL 'Read Attribute' functions, as described in Section 2.3.2.

Parent topic:*Pressure Measurement Cluster*

Pressure Measurement Events There are no events specific to the Pressure Measurement cluster.

Parent topic:*Pressure Measurement Cluster*

Functions The following Pressure Measurement cluster function is provided:

- eCLD_PressureMeasurementCreatePressureMeasurement

eCLD_PressureMeasurementCreatePressureMeasurement

```
teZCL_Status eCLD_PressureMeasurementCreatePressureMeasurement(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Pressure Measurement cluster on an endpoint. The cluster instance is created on the endpoint associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates a Pressure Measurement cluster instance on the endpoint, but instances of other clusters can also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to *Appendix D.*

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in the *ZigBee Devices User Guide (JNUG3131)*.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length is automatically adjusted by the compiler using the following declaration:

```
uint8 au8PressureMeasurementAttributeControlBits
[(sizeof(asCLD_PressureMeasurementClusterAttributeDefinitions) /
    sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Pressure Measurement cluster. This parameter can refer to a pre-filled structure called sCLD_``PressureM``easurement which is provided in the **PressureMeasurement.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_``PressureM``easurement, which defines the attributes of Pressure Measurement cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic:*Pressure Measurement Cluster*

Return codes The Pressure Measurement cluster function uses the ZCL return codes defined in Section 7.2.

Parent topic:*Pressure Measurement Cluster*

Enumerations

'Attribute ID' Enumerations The following structure contains the enumerations used to identify the attributes of the Pressure Measurement cluster.

```
typedef enum
{
 E_CLD_PRESSUREMEAS_ATTR_ID_MEASURED_VALUE   = 0x0000,
 E_CLD_PRESSUREMEAS_ATTR_ID_MIN_MEASURED_VALUE,
 E_CLD_PRESSUREMEAS_ATTR_ID_MAX_MEASURED_VALUE,
 E_CLD_PRESSUREMEAS_ATTR_ID_TOLERANCE,
}teCLD_PM_ClusterID;
```

Parent topic: Enumerations

Parent topic:*Pressure Measurement Cluster*

Structures There are no structures specific to the Pressure Measurement cluster.

Parent topic:*Pressure Measurement Cluster*

Compile-time Options This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the Pressure Measurement cluster.

To enable the Pressure Measurement cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_PRESSURE_MEASUREMENT
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define PRESSURE_MEASUREMENT_SERVER
#define PRESSURE_MEASUREMENT_CLIENT
```

The Pressure Measurement cluster contains macros that may be optionally specified at compile-time by adding one or more of the following lines to the **zcl_options.h** file.

Optional Attributes Add this line to enable the optional Tolerance attribute:

```
#define CLD_PRESSUREMEAS_ATTR_TOLERANCE
```

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_PRESSUREMEAS_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_PRESSUREMEAS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Pressure Measurement Cluster*

Flow Measurement Cluster This chapter outlines the Flow Measurement cluster, which provides an interface for configuring and obtaining flow measurements for a fluid (liquid or gas).

The Flow Measurement cluster has a Cluster ID of 0x0404.

Overview The Flow Measurement cluster provides an interface for configuring and querying devices that perform flow measurements on a fluid (for example, water).

- The server is located on the device which makes the flow measurements
- The client is located on another device and queries the server for measurements

The cluster is enabled by defining CLD_FLOW_MEASUREMENT in the **zcl_options.h** file. Further compile-time options for the Flow Measurement cluster are detailed in Section 27.9.

Parent topic:*Flow Measurement Cluster*

Cluster structure and attributes The structure definition for the Flow Measurement cluster (server) is:

```
typedef struct
{
#ifdef FLOW_MEASUREMENT_SERVER
 zint16  i16MeasuredValue;
 zint16  i16MinMeasuredValue;
 zint16  i16MaxMeasuredValue;
#ifdef CLD_FLOWMEAS_ATTR_TOLERANCE
  zuint16 u16Tolerance;
#endif
#ifdef CLD_FLOWMEAS_ATTR_ATTRIBUTE_REPORTING_STATUS
  zenum8  u8AttributeReportingStatus;
#endif
#endif
 zuint16  u16ClusterRevision;
} tsCLD_FlowMeasurement;
```

where:

'Flow Measurement Information' Attribute Set

- i16MeasuredValue is a mandatory attribute corresponding to 10 times the measured flow, in units of cubic metres per hour (m3/h). The range of possible values is 0x0000 (0 m3/h) to 0xFFFE (representing 6553.4 m3/h). The value 0xFFFF is used to indicate that the measurement was invalid. In practice, the stored value is limited within the range i16MinMeasuredValue to i16MaxMeasuredValue (see below).
- i16MinMeasuredValue is a mandatory attribute representing a lower limit on the value that can be stored in i16MeasuredValue. It is a value in the range 0x0000 to 0x7FFD and, it must be less than the value of the attribute i16MaxMeasuredValue.
- i16MaxMeasuredValue is a mandatory attribute representing an upper limit on the value that can be stored in i16MeasuredValue. It is a value in the range 0x0000 to 0x7FFE, and it must be greater than the value of the attribute i16MinMeasuredValue.
- u16Tolerance is an optional attribute representing the maximum error associated with the measurement stored in i16MeasuredValue. Thus, the true value of the flow is within the range represented by i16MeasuredValue ± u16Tolerance.

Global Attributes

- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Flow Measurement Cluster*

Initialization and Operation The Flow Measurement cluster must be initialized on both the cluster server and client. Initialization can be done using the **eCLD_FlowMeasurementCreateFlowMeasurement()** function, which creates an instance of the Flow Measurement cluster on a local endpoint.

Once the cluster has been initialized, the application on the server should maintain the cluster attributes (see Section 27.2) with the flow measurements made by the local device. The application on a client can remotely read these measured values using the ZCL 'Read Attribute' functions, as described in Section 2.3.2.

Parent topic:*Flow Measurement Cluster*

Flow Measurement Events There are no events specific to the Flow Measurement cluster.

Parent topic:*Flow Measurement Cluster*

Functions The following Flow Measurement cluster function is provided:

eCLD_FlowMeasurementCreateFlowMeasurement

eCLD_FlowMeasurementCreateFlowMeasurement

```
teZCL_Status eCLD_FlowMeasurementCreateFlowMeasurement(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Flow Measurement cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates a Flow Measurement cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to *Appendix D.*

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in the *ZigBee Devices User Guide (JNUG3131).*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length is automatically adjusted by the compiler using the following declaration:

```
uint8 au8FlowMeasurementAttributeControlBits
    [(sizeof(asCLD_FlowMeasurementClusterAttributeDefinitions) /
    sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.

- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Flow Measurement cluster. This parameter can refer to a pre-filled structure called sCLD_``FlowM``easurement which is provided in the **FlowMeasurement.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_``FlowM``easurement, which defines the attributes of Flow Measurement cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic:*Flow Measurement Cluster*

Return codes The Flow Measurement cluster function uses the ZCL return codes defined in Section7.2.

Parent topic:*Flow Measurement Cluster*

Enumerations

'Attribute ID' Enumerations The following structure contains the enumerations used to identify the attributes of the Flow Measurement cluster.

```
typedef enum
{
    E_CLD_FLOWMEAS_ATTR_ID_MEASURED_VALUE  = 0x0000,
    E_CLD_FLOWMEAS_ATTR_ID_MIN_MEASURED_VALUE,
    E_CLD_FLOWMEAS_ATTR_ID_MAX_MEASURED_VALUE,
    E_CLD_FLOWMEAS_ATTR_ID_TOLERANCE,
} teCLD_FlowMeasurement_AttributeID;
```

Parent topic: Enumerations

Parent topic:*Flow Measurement Cluster*

Structures There are no structures specific to the Flow Measurement cluster.

Parent topic:*Flow Measurement Cluster*

Compile-time Options This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the Flow Measurement cluster.

To enable the Flow Measurement cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_FLOW_MEASUREMENT
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define FLOW_MEASUREMENT_SERVER
#define FLOW_MEASUREMENT_CLIENT
```

The Flow Measurement cluster contains macros that may be optionally specified at compile-time by adding one or more of the following lines to the **zcl_options.h** file.

Optional Attributes Add this line to enable the optional Tolerance attribute:

```
#define CLD_FLOWMEAS_ATTR_TOLERANCE
```

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_FLOWMEAS_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_FLOWMEAS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Flow Measurement Cluster*

Relative Humidity Measurement Cluster This chapter describes the Relative Humidity Measurement cluster which describes how to configure and report relative humidity measurement.

The Relative Humidity Measurement cluster has a Cluster ID of 0x0405.

Overview The Relative Humidity Measurement cluster provides an interface to a humidity measuring device, allowing the configuration of relative humidity measuring and the reporting of measurements.

To use the functionality of this cluster, you must include the file **RelativeHumidityMeasurement.h** in your application and enable the cluster by defining CLD_RELATIVE_HUMIDITY_MEASUREMENT in the **zcl_options.h** file.

A Relative Humidity Measurement cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Relative Humidity Measurement cluster are fully detailed in Section 28.6.

Parent topic:*Relative Humidity Measurement Cluster*

RH Measurement Structure and Attributes The structure definition for the Relative Humidity Measurement cluster (server) is:

```
typedef struct
{
    zuint16     u16MeasuredValue;
    zuint16     u16MinMeasuredValue;
    zuint16     u16MaxMeasuredValue;
#ifdef E_CLD_RHMEAS_ATTR_TOLERANCE
    zuint16     u16Tolerance;
#endif
} tsCLD_RelativeHumidityMeasurement;
```

where:

- u1``6MeasuredValue is a mandatory attribute representing the measured relatively humidity as a percentage in steps of 0.01%, as follows:
  - u1``6MeasuredValue = 100 x relative humidity percentage
  - So, for example, 0x197C represents a relative humidity measurement of 65.24%. The possible values are used as follows:
  - 0x0000 to 0x2710 represent relative humidities from 0% to 100%
  - 0x2711 to 0xFFFE are unused values
  - 0xFFFF indicates an invalid measurement
  - This attribute is updated continuously as measurements are made.
- u16MinMeasuredValue is a mandatory attribute specifying the value of the attribute u16MeasuredValue which corresponds to the minimum possible relative humidity that can be measured. Its value must be less than that of the attribute u16MaxMeasuredValue (below). The special value 0xFFFF is used to indicate that the minimum is not defined.
- u16MaxMeasuredValue is a mandatory attribute specifying the value of the attribute u16MeasuredValue which corresponds to the maximum possible relative humidity that can be measured. Its value must be greater than that of the attribute u16MinMeasuredValue (above). The special value 0xFFFF is used to indicate that the maximum is not defined.
- u16Tolerance is an optional attribute which indicates the magnitude of the maximum possible error in the value of the attribute u16MeasuredValue. The true value will be in the range (u16MeasuredValue – u16Tolerance) to (u16MeasuredValue + u16Tolerance).
- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Relative Humidity Measurement Cluster*

Attributes for Default Reporting The following attributes of the Relative Humidity Measurement cluster can be selected for default reporting:

```
u16MeasuredValue
u16Tolerance
```

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic:*Relative Humidity Measurement Cluster*

Functions The following Relative Humidity Measurement cluster function is provided in the NXP implementation of the ZCL:

- eCLD_RelativeHumidityMeasurementCreateRelativeHumidityMeasurement

eCLD_RelativeHumidityMeasurementCreateRelativeHumidityMeasurement

```
teZCL_Status eCLD_RelativeHumidityMeasurementCreateRelativeHumidityMeasurement(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Relative Humidity Measurement cluster on an endpoint. The cluster instance is created on the endpoint associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates a Relative Humidity Measurement cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length is automatically adjusted by the compiler using the following declaration:

```
uint8 au8RelativeHumidityMeasurementAttributeControlBits
  [(sizeof(asCLD_RelativeHumidityMeasurementClusterAttributeDefinitions) /
  sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server

  - FALSE - client

- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Relative Humidity Measurement cluster. This parameter can refer to a pre-filled structure called sCLD_RelativeHumidityMeasurement which is provided in the **RelativeHumidityMeasurement.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_RelativeHumidityMeasurement which defines the attributes of Relative Humidity Measurement cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic:*Relative Humidity Measurement Cluster*

Enumerations

teCLD_RHM_ClusterID The following structure contains the enumerations used to identify the attributes of the Relative Humidity Measurement cluster.

```
typedef enum
{
    E_CLD_RHMEAS_ATTR_ID_MEASURED_VALUE = 0x0000,  /* Mandatory */
    E_CLD_RHMEAS_ATTR_ID_MIN_MEASURED_VALUE,    /* Mandatory */
    E_CLD_RHMEAS_ATTR_ID_MAX_MEASURED_VALUE,    /* Mandatory */
    E_CLD_RHMEAS_ATTR_ID_TOLERANCE,
} teCLD_RHM_ClusterID;
```

Parent topic: Enumerations

Parent topic:*Relative Humidity Measurement Cluster*

Compile-time Options To enable the Relative Humidity Measurement cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_RELATIVE_HUMIDITY_MEASUREMENT
```

In addition, to include the software for a cluster client or server, it is necessary to add one of the following to the same file:

```
#define RELATIVE_HUMIDITY_MEASUREMENT_CLIENT
#define RELATIVE_HUMIDITY_MEASUREMENT_SERVER
```

Optional Attribute Add this line to enable the optional Tolerance attribute:

```
#define CLD_RHMEAS_ATTR_TOLERANCE
```

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_RHMEAS_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_RHMEAS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Relative Humidity Measurement Cluster*

Occupancy Sensing Cluster This chapter describes the Occupancy Sensing cluster which provides an interface to an occupancy sensor.

The Occupancy Sensing cluster has a Cluster ID of 0x0406.

Overview The Occupancy Sensing cluster provides an interface to an occupany sensor, allowing the configuration of sensing and the reporting of status.

To use the functionality of this cluster, you must include the file **OccupancySensing.h** in your application and enable the cluster by defining CLD_OCCUPANCY_SENSING in the **zcl_options.h** file.

An Occupancy Sensing cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Occupancy Sensing cluster are fully detailed in Section 29.6.

The information that can potentially be stored in this cluster is organised into the following attribute sets:

- Occupancy sensor information
- PIR configuration
- Ultrasonic configuration

This cluster has no associated events. The status of an occupancy sensor can be obtained by reading the 'occupancy' attribute (see Section 29.2) which is automatically maintained by the cluster server. The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

Parent topic:*Occupancy Sensing Cluster*

Occupancy Sensing Structure and Attributes The structure definition for the Occupancy Sensing cluster is:

```
typedef struct
{
#ifdef OCCUPANCY_SENSING_SERVER
    zbmap8    u8Occupancy;
    zenum8    eOccupancySensorType;
#ifdef CLD_OS_ATTR_PIR_OCCUPIED_TO_UNOCCUPIED_DELAY
    zuint16 u16PIROccupiedToUnoccupiedDelay;
#endif
#ifdef CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_DELAY
    zuint8   u8PIRUnoccupiedToOccupiedDelay;
#endif
#ifdef CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
    zuint8   u8PIRUnoccupiedToOccupiedThreshold;
#endif
#ifdef CLD_OS_ATTR_ULTRASONIC_OCCUPIED_TO_UNOCCUPIED_DELAY
    zuint16 u16UltrasonicOccupiedToUnoccupiedDelay;
#endif
#ifdef CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_DELAY
    zuint8   u8UltrasonicUnoccupiedToOccupiedDelay;
#endif
#ifdef CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
    zuint8       u8UltrasonicUnoccupiedToOccupiedThreshold;
#endif
#ifdef CLD_OS_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8   u8AttributeReportingStatus;
#endif
#endif
    zuint16 u16ClusterRevision;
} tsCLD_OccupancySensing;
```

where:

'Occupancy Sensor Information' Attribute Set

- u8Occupancy is a mandatory attribute indicating the sensed occupancy in a bitmap in which bit 0 is used as follows (and all other bits are reserved):
  - bit 0 = 1 : occupied
  - bit 0 = 0 : unoccupied
- eOccupancySensorType is a mandatory attribute indicating the type of occupancy sensor, as follows:
  - 0x00 : PIR
  - 0x01 : Ultrasonic
  - 0x02 : PIR and ultrasonic

'PIR Configuration' Attribute Set

- u16P``IROccupiedToUnoccupiedDelay is an optional attribute for a PIR detector representing the time delay, in seconds, between the last detected movement and the sensor changing its occupancy state from 'occupied' to 'unoccupied'
- u8PIRUnoccupiedToOccupiedDelay is an optional attribute for a PIR detector representing the time delay, in seconds, between the detection of movement and the sensor changing its occupancy state from 'unoccupied' to 'occupied'. The interpretation of this attribute changes when it is used in conjunction with the corresponding threshold attribute (see below)

- u8PIRUnoccupiedToOccupiedThreshold is an optional threshold attribute that can be used in conjunction with the delay attribute u8PIRUnoccupiedToOccupiedDelay to allow for false positive detections. Use of this threshold attribute changes the interpretation of the delay attribute. The threshold represents the minimum number of detections required within the delay-period before the sensor will change its occupancy state from 'unoccupied' to 'occupied'. The minimum valid threshold value is 1

'Ultrasonic Configuration' Attribute Set

- u16U``ltrasonicOccupiedToUnoccupiedDelay is an optional attribute for an Ultrasonic detector representing the time delay, in seconds, between the last detected movement and the sensor changing its occupancy state from 'occupied' to 'unoccupied'
- u8UltrasonicUnoccupiedToOccupiedDelay is an optional attribute representing the time delay, in seconds, between the detection of movement and the sensor changing its occupancy state from 'unoccupied' to 'occupied'. The interpretation of this attribute changes when it is used in conjunction with the corresponding threshold attribute (see below)
- u8UltrasonicUnoccupiedToOccupiedThreshold is an optional threshold attribute that can be used in conjunction with the delay attribute u8UltrasonicUnoccupiedToOccupiedDelay to allow for false positive detections. Use of this threshold attribute changes the interpretation of the delay attribute. The threshold represents the minimum number of detections required within the delay-period before the sensor will change its occupancy state from 'unoccupied' to 'occupied'. The minimum valid threshold value is 1

Note: The 'Occupied To Unoccupied' and 'Unoccupied To Occupied' attributes can be used to reduce sensor 'chatter' when an occupancy sensor is deployed in an area in which the occupation frequently changes

Note: (e.g. in a corridor).

Global Attributes

- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.

```
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specifica-
tion on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster re-
vision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This at-
tribute is also described in [Section 2.4](../../ZCL_fundamentals/topics/global_attributes.md#id_8f80e769-
↪8ccd-4b4f-8609-579f4457d527).
```

Parent topic:*Occupancy Sensing Cluster*

Attributes for Default Reporting The following attribute of the Occupancy Sensing cluster can be selected for default reporting:

```
u8Occupancy
```

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for this attribute is described in Appendix B.3.6.

Parent topic:*Occupancy Sensing Cluster*

Functions The following Occupancy Sensing cluster function is provided in the NXP implementation of the ZCL:

- eCLD_OccupancySensingCreateOccupancySensing

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3. The state of the occupancy sensor can be obtained by reading the u8Occupancy attribute in the tsCLD_OccupancySensing structure on the cluster server (see Section 29.2).

eCLD_OccupancySensingCreateOccupancySensing

```
teZCL_Status eCLD_OccupancySensingCreateOccupancySensing(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Occupancy Sensing cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates an Occupancy Sensing cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Occupancy Sensing cluster. The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* : Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Occupancy Sensing cluster. This parameter can refer to a pre-filled structure called sCLD_OccupancySensing which is provided in the **OccupancySensing.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_OccupancySensing which

defines the attributes of Occupancy Sensing cluster. The function initializes the attributes with default values.

- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic:*Occupancy Sensing Cluster*

Enumerations

teCLD_OS_ClusterID The following structure contains the enumeration used to identify the attributes of the Occupancy Sensing cluster.

```
typedef enum
{
  E_CLD_OS_ATTR_ID_OCCUPANCY = 0x0000,  /* Mandatory */
  E_CLD_OS_ATTR_ID_OCCUPANCY_SENSOR_TYPE,  /* Mandatory */
  E_CLD_OS_ATTR_ID_PIR_OCCUPIED_TO_UNOCCUPIED_DELAY = 0x0010,
  E_CLD_OS_ATTR_ID_PIR_UNOCCUPIED_TO_OCCUPIED_DELAY,
  E_CLD_OS_ATTR_ID_PIR_UNOCCUPIED_TO_OCCUPIED_THRESHOLD,
  E_CLD_OS_ATTR_ID_ULTRASONIC_OCCUPIED_TO_UNOCCUPIED_DELAY = 0x0020,
  E_CLD_OS_ATTR_ID_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_DELAY,
  E_CLD_OS_ATTR_ID_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
} teCLD_OS_ClusterID;
```

Parent topic: Enumerations

Parent topic:*Occupancy Sensing Cluster*

Compile-time options To enable the Occupancy Sensing cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_OCCUPANCY_SENSING
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one of the following to the same file:

```
#define OCCUPANCY_SENSING_CLIENT
#define OCCUPANCY_SENSING_SERVER
```

Optional Attributes Add this line to enable the optional PIR Occupied To Unoccupied Delay attribute:

```
#define CLD_OS_ATTR_PIR_OCCUPIED_TO_UNOCCUPIED_DELAY
```

Add this line to enable the optional PIR Unoccupied To Occupied Delay attribute:

```
#define CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_DELAY
```

Add this line to enable the optional PIR Unoccupied To Occupied Threshold attribute:

```
#define CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
```

Add this line to enable the optional Ultrasonic Occupied To Unoccupied Delay attribute:

```
#define CLD_OS_ATTR_ULTRASONIC_OCCUPIED_TO_UNOCCUPIED_DELAY
```

Add this line to enable the optional Ultrasonic Unoccupied To Occupied Delay attribute:

```
#define CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_DELAY
```

Add this line to enable the Ultrasonic PIR Unoccupied To Occupied Threshold attribute:

```
#define CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
```

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_OS_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_OS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Occupancy Sensing Cluster*

Electrical Measurement Cluster This chapter outlines the Electrical Measurement cluster, which provides an interface for obtaining electrical measurements from a device.

The Electrical Measurement cluster has a Cluster ID of 0x0B04.

Overview The Electrical Measurement cluster provides an interface for querying devices for electrical measurements.

- The server is located on the device which makes the electrical measurements
- The client is located on another device and queries the server for measurements

Separate instances of the Electrical Measurement cluster server can be implemented across multiple endpoints within the same physical unit - that is, one server instance per endpoint. An example is a power extension unit containing multiple outlets, where each power outlet allows electrical measurements to be made on the supplied power (e.g. AC RMS voltage and current).

The cluster is enabled by defining CLD_ELECTRICAL_MEASUREMENT in the **zcl_options.h** file. Further compile-time options for the Electrical Measurement cluster are detailed in Section 30.9.

The information that can potentially be stored in this cluster is organised into the following attribute sets:

- Basic Information
- DC Measurement
- DC Formatting
- AC (Non-phase Specific) Measurements
- AC (Non-phase Specific) Formatting
- AC (Single Phase or Phase A) Measurements
- AC Formatting
- DC Manufacturer Threshold Alarms
- AC Manufacturer Threshold Alarms
- AC Phase B Measurements
- AC Phase C Measurements

Note that not all of the above attribute sets are currently implemented in the NXP software and not all attributes within a supported attribute set are implemented (see Section 30.2 for the supported attribute sets and attributes).

Parent topic:*Electrical Measurement Cluster*

Cluster structure and attributes The structure definition for the Electrical Measurement cluster (server) is:

```
typedef struct
{
#ifdef ELECTRICAL_MEASUREMENT_SERVER
zbmap32 u32MeasurementType;
#ifdef CLD_ELECTMEAS_ATTR_AC_FREQUENCY
zuint16 u16ACFrequency;
#endif
#ifdef CLD_ELECTMEAS_ATTR_RMS_VOLTAGE
zuint16 u16RMSVoltage;
#endif
#ifdef CLD_ELECTMEAS_ATTR_RMS_CURRENT
zuint16 u16RMSCurrent;
#endif
#ifdef CLD_ELECTMEAS_ATTR_ACTIVE_POWER
zint16 i16ActivePower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_REACTIVE_POWER
zint16 i16ReactivePower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_APPARENT_POWER
zuint16 u16ApparentPower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_POWER_FACTOR
zint8 i8PowerFactor;
#endif
#ifdef CLD_ELECTMEAS_ATTR_AC_VOLTAGE_MULTIPLIER
zuint16 u16ACVoltageMultiplier;
#endif
#ifdef CLD_ELECTMEAS_ATTR_AC_VOLTAGE_DIVISOR
```

(continues on next page)

(continued from previous page)

```
zuint16 u16ACVoltageDivisor;
#endif
#ifdef CLD_ELECTMEAS_ATTR_AC_CURRENT_MULTIPLIER
zuint16 u16ACCurrentMultiplier;
#endif
#ifdef CLD_ELECTMEAS_ATTR_AC_CURRENT_DIVISOR
zuint16 u16ACCurentDivisor;
#endif
#ifdef CLD_ELECTMEAS_ATTR_AC_POWER_MULTIPLIER
zuint16 u16ACPowerMultiplier;
#endif
#ifdef CLD_ELECTMEAS_ATTR_AC_POWER_DIVISOR
zuint16 u16ACPowerDivisor;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_APPARENT_POWER
zuint32 u32ManSpecificApparentPower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_NON_ACTIVE_POWER
zuint32 u32NonActivePower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_FNDMTL_REACTIVE_POWER
zint32 i32FundamentalReactivePower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_FNDMTL_APPARENT_POWER
zuint32 u32FundamentalApparentPower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_FNDMTL_POWER_FACTOR
zuint16 u16FundamentalPowerFactor;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_NON_FNDMTL_APPARENT_POWER
zuint32 u32NonFundamentalApparentPower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_TOTAL_HARMONIC_DISTORTION
zuint32 u32TotalHarmonicDistortion;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_VBIAS
zuint32 u32VBias;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_DIVISOR
zuint16 u16ManSpecDivisor;
#endif
#endif
zuint16 u16ClusterRevision;
}tsCLD_ElectricalMeasurement;
```

where:

'Basic Information' Attribute Set u32MeasurementType is a mandatory attribute which is a bitmap indicating the types of electrical measurement that can be performed by the deivice on which the cluster server resides. The bitmap is detailed below(a bit is set to '1' if the corresponding measurement type is supported, or to '0' otherwise):

Bits	Measurement Type
0	Active measurement (AC)
1	Reactive measurement (AC)
2	Apparent measurement (AC)
3	Phase A measurement
4	Phase B measurement
5	Phase C measurement
6	DC measurement
7	Harmonics measurement
8	Power quality measurement
9-31	Reserved

'AC (Non-phase Specific) Measurements' Attribute Set

- u16ACFrequency is an optional attribute containing the most recent measurement of the AC frequency, in Hertz(Hz). The special value 0xFFFF is used to indicate that the frequency cannot be measured.

'AC (Single Phase or Phase A) Measurements' Attribute Set Note that the attributes u16RMSVoltage, u16RMSCurrent and i16ActivePower must be enabled in conjunction with the corresponding multiplier/divisor pair in the 'AC Formatting' attribute set.

- u16RMSVoltage is an optional attribute containing the most recent measurement of the Root Mean Square (RMS) voltage, in Volts. The special value 0xFFFF is used to indicate that the RMS voltage cannot be measured. Note that the 'AC Formatting' attributes u16ACVoltageMultiplier and u16ACVoltageDivisor must be implemented with this attribute.
- u16RMSCurrent is an optional attribute containing the most recent measurement of the Root Mean Square (RMS) current, in Amps. The special value 0xFFFF is used to indicate that the RMS current cannot be measured. Note that the 'AC Formatting' attributes u16ACCurrentMultiplier and u16ACCurrentDivisor must be implemented with this attribute.
- i16ActivePower is an optional attribute containing the present single-phase or Phase-A demand for active power, in Watts (W). A positive value represents active power delivered to the premises and a negative value represents active power received from the premises. Note that the 'AC Formatting' attributes u16ACPowerMultiplier and u16ACPowerDivisor must be implemented with this attribute.
- i16ReactivePower is an optional attribute containing the present single-phase or Phase-A demand for reactive power, in Volts-Amps-reactive (VAr). A positive value represents reactive power delivered to the premises and a negative value represents reactive power received from the premises.
- u16ApparentPower is an optional attribute containing the present single-phase or Phase-A demand for apparent power, in Volts-Amps (VA). This value is the positive square-root of i16ActivePower squared plus i16ReactivePower squared.
- i8PowerFactor is an optional attribute containing the single-phase or Phase-A power factor ratio represented as a multiple of 0.01(e.g. the attribute value 0x0C represents a ratio of 0.12).

'AC Formatting' Attribute Set The following attributes come in multiplier/divisor pairs, where each pair corresponds to an attribute of the 'AC (Single Phase or Phase A) Measurements' attribute set and must only be enabled if the corresponding attribute is enabled.

- u16ACVoltageMultiplier is an optional attribute containing the multiplication factor to be applied to the value of the u16RMSVoltage attribute (above). This multiplication factor must be used in conjunction with the u16ACVoltageDivisor division factor. The value 0x0000 is not valid.
- u16ACVoltageDivisor is an optional attribute containing the division factor to be applied to the value of the u16RMSVoltage attribute (above). This division factor must be used in conjunction with the u16ACVoltageMultiplier multiplication factor. The value 0x0000 is not valid.
- u16ACCurrentMultiplier is an optional attribute containing the multiplication factor to be applied to the value of the u16RMSCurrent attribute (above). This multiplication factor must be used in conjunction with the u16ACCurrentDivisor division factor. The value 0x0000 is not valid.
- u16ACCurrentDivisor is an optional attribute containing the division factor to be applied to the value of the u16RMSCurrent attribute (above). This division factor must be used in conjunction with the u16ACCurrentMultiplier multiplication factor. The value 0x0000 is not valid.
- u16ACPowerMultiplier is an optional attribute containing the multiplication factor to be applied to the value of the i16ActivePower attribute (above). This multiplication factor must be used in conjunction with the u16ACPowerDivisor division factor. The value 0x0000 is not valid.
- u16ACPowerDivisor is an optional attribute containing the division factor to be applied to the value of the i16ActivePower attribute (above). This division factor must be used in conjunction with the u16ACPowerMultiplier multiplication factor. The value 0x0000 is not valid.

Manufacturer-specific Attributes

- u32ManSpecificApparentPower is an optional manufacturer-defined attribute containing the demand for apparent power.
- u32NonActivePower is an optional manufacturer-defined attribute containing the demand for non-active power.
- i32FundamentalReactivePower is an optional manufacturer-defined attribute containing the demand for fundamental reactive power.
- u32FundamentalApparentPower is an optional manufacturer-defined attribute containing the demand for fundamental apparent power.
- u16FundamentalPowerFactor is an optional manufacturer-defined attribute representing the power factor of a fundamental power system u32NonFundamentalApparentPower is an optional manufacturer-defined attribute representing the power factor of a non-fundamental (harmonic) power system.
- u32TotalHarmonicDistortion is an optional manufacturer-defined attribute representing the total harmonic distortion present in the delivered power signal.
- u32VBias is an optional manufacturer-defined attribute representing the bias voltage.
- u16ManSpecDivisor is an optional manufacturer-defined attribute representing a power divisor.

Global Attributes

- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attributeis also described in Section2.4.

Parent topic:*Electrical Measurement Cluster*

Initialisation and Operation The Electrical Measurement cluster must be initialised on both the cluster server and client. This can be done using the function **eCLD_ElectricalMeasurementCreateElectricalMeasurement()**, which creates an instance of the Electrical Measurement cluster on a local endpoint.

Once the cluster has been initialized, the application on the server should maintain the cluster attributes (see Section 30.2) with the electrical measurements made by the local device. The application on a client can remotely read these measured values using the ZCL 'Read Attribute' functions, as described in Section 2.3.2.

Parent topic:*Electrical Measurement Cluster*

Electrical Measurement Events There are no events specific to the Electrical Measurement cluster.

Parent topic:*Electrical Measurement Cluster*

Functions The following Electrical Measurement cluster function is provided:

- **Function** **Page**
- eCLD_ElectricalMeasurementCreateElectricalMeasurement 618

eCLD_ElectricalMeasurementCreateElectricalMeasurement

```
teZCL_Status eCLD_ElectricalMeasurementCreateElectricalMeasurement(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Electrical Measurement cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create an Electrical Measurement cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to *Appendix D.*

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in the *ZigBee Devices User Guide (JNUG3131)*.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length is automatically adjusted by the compiler using the following declaration:

```
uint8 au8ElectricalMeasurementAttributeControlBits
[(sizeof(asCLD_ElectricalMeasurementClusterAttributeDefinitions) / sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Electrical Measurement cluster. This parameter can refer to a pre-filled structure called sCLD_``ElectricalMeasurement which is provided in the **ElectricalMeasurement.h** file.
- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_``ElectricalMeasurement which defines the attributes of Electrical Measurement cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic:*Electrical Measurement Cluster*

Return codes The Electrical Measurement cluster function uses the ZCL return codes, listed in Section 7.2.

Parent topic:*Electrical Measurement Cluster*

Enumerations

'Attribute ID' Enumerations The following structure contains the enumerations used to identify the attributes of the Electrical Measurement cluster.

```
typedef enum
{
    E_CLD_ELECTMEAS_ATTR_ID_MEASUREMENT_TYPE
    E_CLD_ELECTMEAS_ATTR_ID_AC_FREQUENCY
    E_CLD_ELECTMEAS_ATTR_ID_RMS_VOLTAGE                = 0x0505,
    E_CLD_ELECTMEAS_ATTR_ID_RMS_CURRENT                = 0x0508,
    E_CLD_ELECTMEAS_ATTR_ID_ACTIVE_POWER               = 0x050B,
    E_CLD_ELECTMEAS_ATTR_ID_REACTIVE_POWER             = 0x050E,
    E_CLD_ELECTMEAS_ATTR_ID_APPARENT_POWER             = 0x050F,
    E_CLD_ELECTMEAS_ATTR_ID_POWER_FACTOR               = 0x0510,
    E_CLD_ELECTMEAS_ATTR_ID_AC_VOLTAGE_MULTIPLIER      = 0x0600,
    E_CLD_ELECTMEAS_ATTR_ID_AC_VOLTAGE_DIVISOR         = 0x0601,
    E_CLD_ELECTMEAS_ATTR_ID_AC_CURRENT_MULTIPLIER      = 0x0602,
```

(continues on next page)

(continued from previous page)

```
    E_CLD_ELECTMEAS_ATTR_ID_AC_CURRENT_DIVISOR          = 0x0603,
    E_CLD_ELECTMEAS_ATTR_ID_AC_POWER_MULTIPLIER         = 0x0604,
    E_CLD_ELECTMEAS_ATTR_ID_AC_POWER_DIVISOR            = 0x0605,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_APPARENT_POWER     = 0xFF00,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_NON_ACTIVE_POWER,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_FNDMTL_REACTIVE_POWER,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_FNDMTL_APPARENT_POWER,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_FNDMTL_POWER_FACTOR,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_NON_FNDMTL_APPARENT_POWER,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_TOTAL_HARMONIC_DISTORTION,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_VBIAS,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_DIVISOR,
} teCLD_ElectricalMeasurement_AttributeID;
```

Parent topic: Enumerations

Parent topic:*Electrical Measurement Cluster*

Structures There are no structures specific to the Electrical Measurement cluster.

Parent topic:*Electrical Measurement Cluster*

Compile-time options This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the Electrical Measurement cluster.

To enable the Electrical Measurement cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_ELECTRICAL_MEASUREMENT
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define ELECTRICAL_MEASUREMENT_SERVER
#define ELECTRICAL_MEASUREMENT_CLIENT
```

Optional Attributes The optional attributes for the Electrical Measurement cluster (see Section 30.2) are enabled by defining:

- CLD_ELECTMEAS_ATTR_AC_FREQUENCY
- CLD_ELECTMEAS_ATTR_RMS_VOLTAGE
- CLD_ELECTMEAS_ATTR_RMS_CURRENT
- CLD_ELECTMEAS_ATTR_ACTIVE_POWER
- CLD_ELECTMEAS_ATTR_REACTIVE_POWER
- CLD_ELECTMEAS_ATTR_APPARENT_POWER
- CLD_ELECTMEAS_ATTR_POWER_FACTOR
- CLD_ELECTMEAS_ATTR_AC_VOLTAGE_MULTIPLIER
- CLD_ELECTMEAS_ATTR_AC_VOLTAGE_DIVISOR
- CLD_ELECTMEAS_ATTR_AC_CURRENT_MULTIPLIER
- CLD_ELECTMEAS_ATTR_AC_CURRENT_DIVISOR

- CLD_ELECTMEAS_ATTR_AC_POWER_MULTIPLIER
- CLD_ELECTMEAS_ATTR_AC_POWER_DIVISOR
- CLD_ELECTMEAS_ATTR_MAN_SPEC_APPARENT_POWER
- CLD_ELECTMEAS_ATTR_MAN_SPEC_NON_ACTIVE_POWER
- CLD_ELECTMEAS_ATTR_MAN_SPEC_FNDMTL_REACTIVE_POWER
- CLD_ELECTMEAS_ATTR_MAN_SPEC_FNDMTL_APPARENT_POWER
- CLD_ELECTMEAS_ATTR_MAN_SPEC_FNDMTL_POWER_FACTOR
- CLD_ELECTMEAS_ATTR_MAN_SPEC_NON_FNDMTL_APPARENT_POWER
- CLD_ELECTMEAS_ATTR_MAN_SPEC_TOTAL_HARMONIC_DISTORTION
- CLD_ELECTMEAS_ATTR_MAN_SPEC_VBIAS
- CLD_ELECTMEAS_ATTR_MAN_SPEC_DIVISOR

Global Attributes Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_APPLIANCE_STATISTICS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Electrical Measurement Cluster*

Part V: Lighting Clusters

- This part comprises two chapters:
  - *Chapter 31* details the **Colour Control** cluster
  - *Chapter 32* details the **Ballast Configuration** cluster

Colour Control Cluster This chapter describes the Colour Control cluster which is defined in the ZCL.

The Colour Control cluster has a Cluster ID of 0x0300.

Overview The Colour Control cluster is used to control the colour of a light.

Note: Note 1: This cluster should normally be used with the Level Control cluster (see *Chapter 16*) and On/Off cluster (see *Chapter 14*). This is assumed to be the case in this description.

Note: Note 2: This cluster only controls the colour balance and not the overall brightness of a light. The brightness is adjusted using the Level Control cluster.

The Colour Control cluster provides the facility to specify the colour of a light in the colour space defined in the *Commission Internationale de l'Éclairage (CIE) specification (1931)*. Colour control can be performed in terms of any of the following:

- x and y values, as defined in the CIE specification
- hue and saturation
- colour temperature

To use the functionality of this cluster, you must include the file **ColourControl.h** in your application and enable the cluster by defining CLD_COLOUR_CONTROL in the **zcl_options.h** file - see Section 31.9.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to change the colour on the local light device.
- The cluster client is able to send commands to change the colour on the remote light device.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Colour Control cluster are fully detailed in Section31.9.

The information that can potentially be stored in this cluster is organized into the following attribute sets:

- Colour Information
- Defined Primaries Information
- Additional Defined Primaries Information
- Defined Colour Point Settings
- Enhanced Colour Mode

Parent topic:*Colour Control Cluster*

Colour Control Cluster structure and attributes The structure definition for the Colour Control cluster is:

```
typedef struct
{
#ifdef COLOUR_CONTROL_SERVER
    /* Colour information attribute set */
#ifdef CLD_COLOURCONTROL_ATTR_CURRENT_HUE
    zuint8              u8CurrentHue;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_CURRENT_SATURATION
    zuint8              u8CurrentSaturation;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_REMAINING_TIME
    zuint16             u16RemainingTime;
#endif
    zuint16               u16CurrentX;
    zuint16              u16CurrentY;
#ifdef CLD_COLOURCONTROL_ATTR_DRIFT_COMPENSATION
    zenum8               u8DriftCompensation;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COMPENSATION_TEXT
    tsZCL_CharacterString   sCompensationText;
    uint8            au8CompensationText[
        CLD_COLOURCONTROL_COMPENSATION_TEXT_MAX_STRING_LENGTH];
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIRED
    zuint16             16ColourTemperatureMired;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_MODE
    zenum8                u8ColourMode;
#endif
    zbmap8                u8Options;
    /* Defined Primaries Information attribute set */
```

(continues on next page)

(continued from previous page)

```
#ifdef CLD_COLOURCONTROL_ATTR_NUMBER_OF_PRIMARIES
    zuint8              u8NumberOfPrimaries;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_1_X
    zuint16             u16Primary1X;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_1_Y
    zuint16             u16Primary1Y;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_1_INTENSITY
    zuint8              u8Primary1Intensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_2_X
    zuint16             u16Primary2X;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_2_Y
    zuint16             u16Primary2Y;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_2_INTENSITY
    zuint8              u8Primary2Intensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_3_X
    zuint16             u16Primary3X;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_3_Y
    zuint16             u16Primary3Y;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_3_INTENSITY
    zuint8              u8Primary3Intensity;
#endif
    /* Additional Defined Primaries Information attribute set */
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_4_X
    zuint16             u16Primary4X;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_4_Y
    zuint16             u16Primary4Y;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_4_INTENSITY
    zuint8              u8Primary4Intensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_5_X
    zuint16             u16Primary5X;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_5_Y
    zuint16             u16Primary5Y;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_5_INTENSITY
    zuint8              u8Primary5Intensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_6_X
    zuint16             u16Primary6X;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_6_Y
    zuint16             u16Primary6Y;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_6_INTENSITY
    zuint8              u8Primary6Intensity;
#endif
    /* Defined Colour Points Settings attribute set */
#ifdef CLD_COLOURCONTROL_ATTR_WHITE_POINT_X
    zuint16             u16WhitePointX;
```

(continues on next page)

(continued from previous page)

```
#endif
#ifdef CLD_COLOURCONTROL_ATTR_WHITE_POINT_Y
    zuint16                u16WhitePointY;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_X
    zuint16                u16ColourPointRX;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_Y
    zuint16                u16ColourPointRY;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_INTENSITY
    zuint8              u8ColourPointRIntensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_X
    zuint16             u16ColourPointGX;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_Y
    zuint16                u16ColourPointGY;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_INTENSITY
    zuint8              u8ColourPointGIntensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_X
    zuint16                u16ColourPointBX;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_Y
    zuint16                u16ColourPointBY;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_INTENSITY
    zuint8              u8ColourPointBIntensity;
#endif
/* Colour information attribute set */
#ifdef CLD_COLOURCONTROL_ATTR_ENHANCED_CURRENT_HUE
    zuint16                u16EnhancedCurrentHue;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_ENHANCED_COLOUR_MODE
    zenum8                 u8EnhancedColourMode;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_ACTIVE
    zuint8              u8ColourLoopActive;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_DIRECTION
    zuint8              u8ColourLoopDirection;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_TIME
    zuint16             u16ColourLoopTime;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_START_ENHANCED_HUE
    zuint16             u16ColourLoopStartEnhancedHue;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_STORED_ENHANCED_HUE
    zuint16                 u16ColourLoopStoredEnhancedHue;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_CAPABILITIES
    zuint16                 u16ColourCapabilities;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIRED_PHY_MIN
    zuint16            u16ColourTemperatureMiredPhyMin;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIRED_PHY_MAX
    zuint16           u16ColourTemperatureMiredPhyMax;
```

(continues on next page)

(continued from previous page)

```
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COUPLE_COLOUR_TEMPERATURE_TO_LEVEL_MIN_
↪MIRED
    zuint16     u16CoupleColourTempToLevelMinMired;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_STARTUP_COLOUR_TEMPERATURE_MIRED
    zuint16          u16StartupColourTemperatureMired;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8           u8AttributeReportingStatus;
#endif
#endif
    zuint16          u16ClusterRevision;
} tsCLD_ColourControl;
```

where:

'Colour Information' Attribute Set Note that the attributes u8CurrentHue, u8CurrentSaturation, u16CurrentX,``u16CurrentY and u16ColourTemperatureMired are enabled as part of 'Colour Capabilities' groups - see Table 34 on page 704.

- u8CurrentHue is the current hue value of the light in the range 0-254. This value can be converted to hue in degrees using the following formula:

  hue = u8CurrentHue x 360/254. This attribute is only valid when the attributes u8CurrentSaturation and u8ColorMode are also implemented.

- u8CurrentSaturation is the current saturation value of the light in the range 0-254. This value can be converted to saturation as a fraction using the following formula: saturation = u8CurrentSaturation/254. This attribute is only valid when the attributes u8CurrentHue and u8ColorMode are also implemented.

- u16RemainingTime is the time duration, in tenths of a second, before the currently active command completes.

- u16CurrentX is the current value for the chromaticity x, as defined in the CIE xyY colour space, in the range 0-65279. The normal value of x is calculated using the following formula: x = u16CurrentX/65536.

- u16CurrentY is the current value for the chromaticity y, as defined in the CIE xyY colour space, in the range 0-65279. The normal value of y is calculated using the following formula: y = u16CurrentY/65536.

- u8DriftCompensation indicates the mechanism, if any, is being used to compensate for colour/intensity drift over time. One of the following values is specified:

u8DriftCompensation	Drift Compensation Mechanism
0x00	None
0x01	Other or unknown
0x02	Temperature monitoring
0x03	Optical luminance monitoring and feedback
0x04	Optical colour monitoring and feedback
0x05 - 0xFF	Reserved

- The following optional pair of attributes are used to store a textual indication of the drift compensation mechanism used:
  - sCompensationText is a tsZCL_CharacterString structure (see Section 6.1.14) for a character string representing the drift compensation method used

  - au8CompensationText[] is a byte-array which contains the character data bytes representing the drift compensation method used

- u16ColourTemperatureMired is the colour temperature of the light expressed as a micro reciprocal degree (mired) value. It is a scaled reciprocal of the current value of the colour temperature, in the range 1-65279

  (0 is undefined and 65535 indicates an invalid value). The colour temperature, in Kelvin, is calculated using the following formula:

  T = 1000000/u16ColourTemperature``Mired. This attribute is only valid when the attribute u8Colo``u``rMode is also implemented.

- u8ColourMode indicates which method is currently being used to control the colour of the light. One of the following values is specified:

u8ColourMode	Colour Control Method/Attributes
0x00	Hue and saturation

(u8CurrentHue and u8CurrentSaturation)

| |0x01|Chromaticities x and y from CIE xyY colour space

(u16CurrentX and u16CurrentY)

| |0x02|Colour temperature (u16ColourTemperature``Mired)| |0x03 - 0xFF|Reserved|

- u8Options is a bitmap which allows behaviors connected with certain commands to be defined (these behaviors should only be defined during commissioning), as follows:

Bits	Name	Description
0	Change	Defines whether changes to the Colour Control cluster can be made from control clusters (e.g. Level Control) when the bOnOff attribute of the On/Off cluster is zero (off):

- 1 – Allow changes
- 0 – Do not allow changes

| |1-7|-|Reserved|

'Defined Primaries Information'

- u8NumberOfPrimaries is the number of colour primaries implemented on the device, in the range 1-6 (0xFF is used if the number of primaries is unknown).
  - For each colour primary, there is a set of three attributes used (see below) - for example, for the first primary this attribute trio comprises u16Primary1X, u16Primary1Y and u8Primary1Intensity. Therefore, the number of primaries specified determines the number of these attribute trios used.

Note: The number of primaries is set using a macro at compile-time (see Section 31.9). This automatically enables the relevant u16PrimaryNX, u16PrimaryNY and u8PrimaryNIntensity (N=1 to 6) attributes.

- The attribute definitions below are valid for colour primary N, where N is 1, 2 or 3.
- u16PrimaryNX is the value for the chromaticity x for colour primary N, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of x is calculated using the following formula: x = u16PrimaryNX/65536.

- u16PrimaryNY is the value for the chromaticity y for colour primary N, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of y is calculated using the following formula: y = u16PrimaryNY/65536.
- u8PrimaryNIntensity is a representation of the maximum intensity of colour primary 1, normalized such that the primary with the highest maximum intensity has the value 0xFE.

'Additional Defined Primaries Information' Attribute Set

- The attribute definitions for this set are as for u16PrimaryNX, u16PrimaryNY and u8PrimaryNIntensity above, where N is 4, 5 or 6.
- As indicated in the Note above for the *'Defined Primaries Information' Attribute*Set, these attributes are enabled automatically according to the number of required primaries defined at compile-time (see Section 31.9).

'Defined Colour Points Settings'

- u16WhitePointX is the value for the chromaticity x for the white point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of x is calculated using the following formula:

  x = u16WhitePointX/65536.
- u16WhitePointY is the value for the chromaticity y for the white point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of y is calculated using the following formula:

  y = u16WhitePointY/65536.
- u16ColourPointRX is the value for the chromaticity x for the red colour point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of x is calculated using the following formula:

  x = u16ColourPointRX/65536.
- u16ColourPointRY is the value for the chromaticity y for the red colour point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of y is calculated using the following formula:

  y = u16ColourPointRY/65536.
- u8ColourPointRIntensity is a representation of the relative intensity of the red colour point of the device, normalized such that the colour point with the highest relative intensity has the value 0xFE (the value 0xFF indicates an invalid value).
- u16ColourPointGX is the value for the chromaticity x for the green colour point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of x is calculated using the following formula:

  x = u16ColourPointGX/65536.
- u16ColourPointGY is the value for the chromaticity y for the green colour point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of y is calculated using the following formula:

  y = u16ColourPointGY/65536.
- u8ColourPointGIntensity is a representation of the relative intensity of the green colour point of the device, normalized such that the colour point with the highest relative intensity has the value 0xFE (the value 0xFF indicates an invalid value).
- u16ColourPointBX is the value for the chromaticity x for the blue colour point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of x is calculated using the following formula:

x = u16ColourPointBX/65536.

- u16ColourPointBY is the value for the chromaticity y for the blue colour point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of y is calculated using the following formula:

  y = u16ColourPointBY/65536.

- u8ColourPointBIntensity is a representation of the relative intensity of the blue colour point of the device, normalized such that the colour point with the highest relative intensity has the value 0xFE (the value 0xFF indicates an invalid value).

Enhanced Colour Mode Attributes These attributes are enabled as part of 'Colour Capabilities' groups - see Table 34 onpage 704.

- u16EnhancedCurrentHue contains the current hue of the light in terms of (unequal) steps around the CIE colour 'triangle':
  - 8 most significant bits represent an index into the XY look-up table that contains the step values, thus indicating the current step used
  - 8 least significant bits represent a linear interpolation value between the current step and next step (up), facilitating a colour zoom
    - The value of the u8CurrentHue attribute is calculated from the above values.
- u8EnhancedColourMode indicates which method is currently being used to control the colour of the light. One of the following values is specified:

u8ColourMode	Colour Control Method/Attributes
0x00	Current hue and current saturation

(u8CurrentHue and u8CurrentSaturation)

| |0x01|Chromaticities x and y from CIE xyY colour space

(u16CurrentX and u16CurrentY)

| |0x02|Colour temperature (u16ColourTemperature``Mired)| |0x03|Enhanced hue and current saturation

(u16EnhancedCurrentHue and u8CurrentSaturation)

| |0x03 - 0xFF|Reserved|

- u8ColourLoopActive indicates whether the colour loop is currently active: 0x01 - active, 0x00 - not active (all other values are reserved). The colour loop follows the hue steps around the CIE colour 'triangle' by incrementing or decrementing the value of u16EnhancedCurrentHue.
- u8ColourLoopDirection indicates the current direction of the colour loop in terms of the direction of change of u16EnhancedCurrentHue:

  0x01 - incrementing, 0x00 - decrementing (all other values are reserved).

- u16ColourLoopTime is the period, in seconds, of a full colour loop - that is, the time to cycle all possible values of u16EnhancedCurrentHue.
- u16ColourLoopStartEnhancedHue indicates the value of u16EnhancedCurrentHue at which the colour loop must be started.
- u16ColourLoopStoredEnhancedHue contains the value of u16EnhancedCurrentHue at which the last colour loop completed (this value is stored on completing a colour loop).
- u16ColourCapabilities is a bitmap indicating the Colour Control cluster features (and attributes) supported by the device, as detailed below (a bit is set to '1' if the feature is supported or '0' otherwise):

Bits	Feature	Attributes
0	Hue/Saturation	“‘
u8CurrentHue		
u8CurrentSaturation		

```
|
|1|Enhanced Hue

 \(Hue/Saturation must also be supported\)

|```
u16EnhancedCurrentHue
```

| |2|Colour Loop

(Enhanced Hue must also be supported)

|“‘ u8ColourLoopActive u8ColourLoopDirection u16ColourLoopTime u16ColourLoopStartEnhancedHue u16ColourLoopStoredEnhancedHue u16ColourCapabilities

```
|
|3|CIE XY Values|```
u16CurrentX
u16CurrentY
```

| |4|Colour Temperature (Mired)|“‘ u16ColourTemperatureMired u16ColourTemperatureMiredPhyMin u16ColourTemperatureMiredPhyMax

```
|
|5-15|Reserved|-|

Macros are provided to select the required Colour Capabilities at compile-time - see [Table 34 on page
↪704](compile-time_options.md#id_d6675cb0-6090-4409-b15f-2c832f964ed4).

-  `u16ColourTemperature``Mired``PhyMin` indicates the minimum value \(supported by the hardware\)
↪of the mired colour temperature attribute.
-  `u16ColourTemperature``Mired``PhyMax` indicates the maximum value \(supported by the hardware\)
↪of the mired colour temperature attribute.
-  `u16CoupleColourTempToLevelMinMired` is an optional attribute that is used when the
↪`u16ColourTemperatureMired` attribute is coupled to the `u8CurrentLevel` attribute of the Level Control
↪cluster \(this is the case when the CoupleColorTempToLevel bit of the `u8Options` attribute of the Level
↪Control cluster is equal to 1\). `u16CoupleColourTempToLevelMinMired` specifies a lower bound on the
↪value of the `u16ColourTemperatureMired` attribute, where this lower bound corresponds to a
↪`u8CurrentLevel` value of 0xFE \(100%\). Note that because the colour temperature is represented as a
↪mired \(reciprocal\) value, a high value of `u8CurrentLevel` corresponds to a low value of
↪`u16ColourTemperatureMired` and the `16CoupleColourTempToLevelMinMired` attribute corresponds to
↪an upper bound on the value of the colour temperature supported by the device. The value of this attribute
↪must be at least equal to the value of `u16ColourTemperatureMiredPhyMin`.
-  `u16StartupColourTemperatureMired` is an optional attribute to define the required start-up colour
↪temperature of a light when it is supplied with power. It determines the initial value of
↪`u16ColourTemperatureMired` on start-up.

## Global Attributes
```

(continues on next page)

(continued from previous page)

```
- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is␣
↪used for the cluster \(see [Section 2.3.5](../../ZCL_fundamentals/topics/attribute_reporting.md#id_
↪d3e2d682-d5e6-4aa5-870f-4d2e0320637e)\). The value of this attribute indicates whether there are attribute␣
↪reports still pending \(0x00\) or the attribute reports are complete \(0x01\) - all other values are reserved.␣
↪This attribute is also described in [Section 2.4](../../ZCL_fundamentals/topics/global_attributes.md#id_
↪8f80e769-8ccd-4b4f-8609-579f4457d527).


u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which␣
↪this cluster instance is based. The cluster specification in the ZCLr6 corresponds to a cluster revision of 1.␣
↪The value is incremented by one for each subsequent revision of the cluster specification. This attribute is␣
↪also described in[Section2.4](../../ZCL_fundamentals/topics/global_attributes.md#id_8f80e769-8ccd-4b4f-
↪8609-579f4457d527).

**Parent topic:**[Colour Control Cluster](../../Colour_Control_cluster/topics/colour_control_cluster.md)
```

Attributes for Default Reporting The following attributes of the Colour Control cluster can be selected for default reporting:

```
u8CurrentHue
u8CurrentSaturation
u16CurrentX
```

- u16CurrentY

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic:*Colour Control Cluster*

Initialization The function **eCLD_ColourControlCreateColourControl()** is used to create an instance of the Colour Control cluster. The function is generally called by the initialization function for the host device.

Parent topic:*Colour Control Cluster*

Sending Commands The NXP implementation of the ZCL provides functions for sending commands between a Colour Control cluster client and server. A command is sent from the client to one or more endpoints on the server. Multiple endpoints can usually be targeted using binding or group addressing.

Note: Any 'Move to', 'Move' or 'Step' command that is currently in progress can be stopped at any time by calling the function: **eCLD_ColourControlCommandStopMoveStepCommandSend()**

Controlling Hue Colour can be controlled in terms of hue, which is related to the dominant wavelength (or frequency) of the light emitted by a lighting device. On a device that supports the Colour Control cluster, the hue is controlled by means of the 'current hue' attribute (u8CurrentHue) of the cluster. This attribute can take a value in the range 0-254, which can be converted to hue in degrees using the following formula:

Hue in degrees = u8CurrentHue x 360/254

The 'current hue' attribute can be controlled in a number of ways using commands of the Colour Control cluster. API functions are available to send these commands to endpoints on remote devices.

'Move to Hue' Command The 'Move to Hue' command allows the 'current hue' attribute to be moved (increased or decreased) to a specified target value in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveToHueCommandSend()

Since the possible hues are represented on a closed boundary, the target hue can be reached by moving the attribute value in either direction, up or down (the attribute value wraps around). Options are also provided for taking the 'shortest route' and 'longest route' around the boundary.

'Move Hue' Command The 'Move Hue' command allows the 'current hue' attribute to be moved in a given direction (increased or decreased) at a specified rate indefinitely, until stopped. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveHueCommandSend()

Since the possible hues are represented on a closed boundary, the movement is cyclic (the attribute value wraps around). The above function can also be used to stop the movement.

'Step Hue' Command The 'Step Hue' command allows the 'current hue' attribute to be moved (increased or decreased) by a specified amount in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandStepHueCommandSend()

Note: Hue can also be moved in conjunction with saturation, as described in Section 31.5.7. The 'enhanced' hue can be moved in similar ways, as described in Section 31.5.5.

Parent topic: Sending Commands

Controlling Saturation Colour can be controlled in terms of saturation, which is related to the spread of wavelengths (or frequencies) in the light emitted by a lighting device. On a device that supports the Colour Control cluster, the saturation is controlled by means of the 'current saturation' attribute (u8CurrentSaturation) of the cluster. This attribute can take a value in the range 0-254, which can be converted to saturation as a fraction using the following formula:

Saturation = u8CurrentSaturation/254

The 'current saturation' attribute can be controlled in a number of ways using commands of the Colour Control cluster. API functions are available to send these commands to endpoints on remote devices.

'Move to Saturation' Command The 'Move to Saturation' command allows the 'current saturation' attribute to be moved (increased or decreased) to a specified target value in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveToSaturationCommandSend()

'Move Saturation' Command The 'Move Saturation' command allows the 'current saturation' attribute to be moved in a given direction (increased or decreased) at a specified rate until stopped or until the current saturation reaches its minimum or maximum value. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveSaturationCommandSend()

The above function can also be used to stop the movement.

'Step Saturation' Command The 'Step Saturation' command allows the 'current saturation' attribute to be moved (increased or decreased) by a specified amount in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandStepSaturationCommandSend()

Note: Saturation can also be moved in conjunction with hue, as described in Section 31.5.7.

Parent topic: Sending Commands

Controlling Colour (CIE x and y Chromaticities) Colour can be controlled in terms of the x and y chromaticities defined in the CIE xyY colour space. On a device that supports the Colour Control cluster, these values are controlled by means of the 'current x' attribute (u16CurrentX) and 'current y' attribute (u16CurrentY) of the cluster. Each of these attributes can take a value in the range 0-65279. The normalized x and y chromaticities can then be calculated from these values using the following formulae:

x = u16CurrentX/65536

y = u16CurrentY/65536

The x and y chromaticity attributes can be controlled in a number of ways using commands of the Colour Control cluster. API functions are available to send these commands to endpoints on remote devices.

'Move to Colour' Command The 'Move to Colour' command allows the 'current x' and 'current y' attributes to be moved (increased or decreased) to specified target values in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveToColourCommandSend()

'Move Colour' Command The 'Move Colour' command allows the 'current x' and 'current y' attributes to be moved in a given direction (increased or decreased) at specified rates until stopped or until both attributes reach their minimum or maximum value. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveColourCommandSend()

The above function can also be used to stop the movement.

'Step Colour' Command The 'Step Colour' command allows the 'current x' and 'current y' attributes to be moved (increased or decreased) by specified amounts in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandStepColourCommandSend()

Parent topic: Sending Commands

Controlling Colour Temperature Colour can be controlled in terms of colour temperature, which is the temperature of an ideal black body which radiates light of a similar hue to that of the lighting device. On a device that supports the Colour Control cluster, the colour temperature is controlled by means of the 'mired colour temperature' attribute (u16ColourTemperature``Mired) of the cluster. This attribute stores a micro reciprocal degree (mired) value, which is a scaled reciprocal of the current value of the colour temperature of the light, in the range 1-65279. The colour temperature, in Kelvin, can be calculated from the attribute value using the following formula:

T = 1000000/u16ColourTemperature``Mired

Note: The movement of colour temperature through colour space always follows the 'Black Body Line'.

'Move to Colour Temperature' Command The 'Move to Colour Temperature' command allows the 'mired colour temperature' attribute to be moved (increased or decreased) to a specified target value in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveToColourTemperatureCommandSend()

'Move Colour Temperature' Command The 'Move Colour Temperature' command allows the 'mired colour temperature' attribute to be moved in a given direction (increased or decreased) at a specified rate until stopped. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveColourTemperatureCommandSend()

The above function can also be used to stop the movement.

Maximum and minimum values for the 'mired colour temperature' attribute during the movement are also specified. If the attribute value reaches the specified maximum or minimum before the required change has been achieved, the movement will automatically stop.

'Step Colour Temperature' Command The 'Step Colour Temperature' command allows the 'mired colour temperature' attribute to be moved (increased or decreased) by a specified amount in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandStepColourTemperatureCommandSend()

Maximum and minimum values for the 'mired colour temperature' attribute during the movement are also specified. If the attribute value reaches the specified maximum or minimum before the required change has been achieved, the movement will automatically stop.

Parent topic: Sending Commands

Controlling 'Enhanced' Hue Colour can be controlled in terms of hue, which is related to the dominant wavelength (or frequency) of the light emitted by a lighting device. The hue can alternatively be controlled by means of the 'enhanced current hue' attribute (u16EnhancedCurrentHue), instead of the 'current hue' attribute (the 'current hue' attribute is automatically adjusted when the 'enhanced current hue' attribute value changes).

The 'enhanced current hue' attribute allows hue to be controlled on a finer scale than the 'current hue' attribute. Hue steps are defined in a look-up table and values between the steps can be achieved through linear interpolation. This 16-bit attribute value therefore comprises two 8-bit components, as described below.

Bits 15-8	Bits 7-0
Index into the look-up table that	

contains the hue step values, thus indicating the current step used

|Linear interpolation value between the

current step and next step (up)

|

Thus, if the current hue step value is *Hi* (where *i* is the relevant table index) and the linear interpolation value is *interp*, the 'enhanced' hue is given by the formula:

Enhanced hue = *Hi* + (*interp*/255) x (*Hi+1* - *Hi*)

To convert this hue to a value in degrees, it is then necessary to multiply by 360/255.

The 'enhanced current hue' attribute can be controlled in a number of ways using commands of the Colour Control cluster. API functions are available to send these commands to endpoints on remote devices.

Note: **Note:**These commands are issued by a cluster client and are performed on a cluster server. The look-up table is user-defined on the server. When this command is received by the server, the user-defined callback function that is invoked must read the entry with the specified index from the look-up table and calculate the corresponding 'enhanced' hue value.

'Enhanced Move to Hue' Command The 'Enhanced Move to Hue' command allows the 'enhanced current hue' attribute to be moved (increased or decreased) to a specified target value in a continuous manner over a specified transition time (the 'current hue' attribute is also moved to a value based on the target 'enhanced current hue' value). This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandEnhancedMoveToHueCommandSend()

Since the possible hues are represented on a closed boundary, the target hue can be reached by moving the attribute value in either direction, up or down (the attribute value wraps around). Options are also provided for taking the 'shortest route' and 'longest route' around the boundary.

'Enhanced Move Hue' Command The 'Enhanced Move Hue' command allows the 'enhanced current hue' attribute to be moved in a given direction (increased or decreased) at a specified rate indefinitely, until stopped (the 'current hue' attribute is also moved through values based on the 'enhanced current hue' value). This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandEnhancedMoveHueCommandSend()

The above function can also be used to stop the movement.

Since the possible hues are represented on a closed boundary, the movement is cyclic (the attribute value wraps around). The above function can also be used to stop the movement.

'Enhanced Step Hue' Command The 'Enhanced Step Hue' command allows the 'enhanced current hue' attribute to be moved (increased or decreased) by a specified amount in a continuous manner over a specified transition time (the 'current hue' attribute is also moved through values based on the 'enhanced current hue' value). This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandEnhancedStepHueCommandSend()

Note: Note 1: 'Enhanced' hue can also be moved in conjunction with saturation, as described in Section31.5.7.

Note: Note 2: The value of the 'enhanced current hue' attribute can be moved around a colour loop, as described in Section 31.5.6.

Parent topic: Sending Commands

Controlling a Colour Loop The colour of a device can be controlled by moving the value of the 'enhanced current hue' attribute around a colour loop corresponding to the CIE colour 'triangle' - refer to Section 31.5.5 for details of the 'enhanced current hue' attribute.

Movement along the colour loop can be controlled using the 'Colour Loop Set' command of the Colour Control cluster. A function is available to send this command to endpoints on remote devices.

'Colour Loop Set' Command The 'Colour Loop Set' command allows movement of the 'enhanced current hue' attribute value around the colour loop to be configured and started. The direction(up or down), start 'enhanced' hue and duration of the movement can be specified. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandColourLoopSetCommandSend()

The above function can also be used to stop the movement.

Parent topic: Sending Commands

Controlling Hue and Saturation Colour can be completely specified in terms of hue and saturation, which respectively represent the dominant wavelength (or frequency) of the light and the spread of wavelengths (around the former) within the light. Therefore, the Colour Control cluster provides commands to change both the hue and saturation at the same time. In fact, commands are provided to control the values of the:

- 'current hue' and 'current saturation' attributes
- 'enhanced current hue' and 'current saturation' attributes

API functions are available to send these commands to endpoints on remote devices.

'Move to Hue and Saturation' Command The 'Move to Hue and Saturation' command allows the 'current hue' and 'current saturation'attributes to be moved to specified target values in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveToHueCommandSend()

'Enhanced Move to Hue and Saturation' Command The 'Enhanced Move to Hue and Saturation' command allows the 'enhanced current hue' and 'current saturation'attributes to be moved to specified target values in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandEnhancedMoveToHueAndSaturationCommandSend()

Parent topic: Sending Commands

Parent topic:*Colour Control Cluster*

Functions The following Colour Control cluster functions are provided in the NXP implementation of the ZCL:

- eCLD_ColourControlCreateColourControl
- eCLD_ColourControlCommandMoveToHueCommandSend
- eCLD_ColourControlCommandMoveHueCommandSend
- eCLD_ColourControlCommandStepHueCommandSend
- eCLD_ColourControlCommandMoveToSaturationCommandSend

- eCLD_ColourControlCommandMoveSaturationCommandSend
- eCLD_ColourControlCommandStepSaturationCommandSend
- eCLD_ColourControlCommandMoveToHueAndSaturationCommandSend
- eCLD_ColourControlCommandMoveToColourCommandSend
- eCLD_ColourControlCommandMoveColourCommandSend
- eCLD_ColourControlCommandStepColourCommandSend
- eCLD_ColourControlCommandEnhancedMoveToHueCommandSend
- eCLD_ColourControlCommandEnhancedMoveHueCommandSend
- eCLD_ColourControlCommandEnhancedStepHueCommandSend
- eCLD_ColourControlCommandEnhancedMoveToHueAndSaturationCommandSend
- eCLD_ColourControlCommandColourLoopSetCommandSend
- eCLD_ColourControlCommandStopMoveStepCommandSend
- eCLD_ColourControlCommandMoveToColourTemperatureCommandSend
- eCLD_ColourControlCommandMoveColourTemperatureCommandSend
- eCLD_ColourControlCommandStepColourTemperatureCommandSend
- eCLD_ColourControl_GetRGB

eCLD_ColourControlCreateColourControl

```
teZCL_Status eCLD_ColourControlCreateColourControl(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits,
    tsCLD_ColourControlCustomDataStructure
    *psCustomDataStructure);
```

Description This function creates an instance of the Colour Control cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates a Colour Control cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be the first Colour Control cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Colour Control cluster. The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* : Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Colour Control cluster. This parameter can refer to a pre-filled structure called sCLD_ColourControl which is provided in the **ColourControl.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_ColourControl which defines the attributes of Colour Control cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 31.7.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandMoveToHueCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveToHueCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_MoveToHueCommandPayload
    *psPayload);
```

Description This function sends a Move to Hue command to instruct a device to move its 'current hue' attribute to a target hue value in a continuous manner within a given time. The hue value, direction and transition time are specified in the payload of the command (see Section 31.7.2).

Since the possible hues are represented on a closed boundary, the target hue can be reached by moving the attribute value in either direction, up or down (the attribute value wraps around). Options are also provided for 'shortest route' and 'longest route' around the boundary.

The device receiving this message generates a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'hue and saturation' mode is selected by setting the 'colour mode' attribute to 0x00, if required. It can then move the 'current hue' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'current hue' attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

This function invokes the ZigBee PRO stack function to transmit the data. In case an error is returned, call the **eZCL_GetLastZpsError()** function to get the error.

Parent topic: Functions

eCLD_ColourControlCommandMoveHueCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveHueCommandSend(
uint8 u8SourceEndPointId,
uint8 u8DestinationEndPointId,
tsZCL_Address *psDestinationAddress,
```

(continues on next page)

(continued from previous page)

```
uint8 *pu8TransactionSequenceNumber,
tsCLD_ColourControl_MoveHueCommandPayload
*psPayload);
```

Description This function sends a Move Hue command to instruct a device to move its 'current hue' attribute value in a given direction at a specified rate for an indefinite time. The direction and rate are specified in the payload of the command (see Section 31.7.2).

The command can request that the hue is moved up or down, or that existing movement is stopped. Since the possible hues are represented on a closed boundary, the movement is cyclic (the attribute value wraps around). Once started, the movement will continue until it is stopped.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'hue and saturation' mode is selected by setting the 'colour mode' attribute to 0x00, if required. It can then move the 'current hue' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'current hue' attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandStepHueCommandSend

```
teZCL_Status eCLD_ColourControlCommandStepHueCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ColourControl_StepHueCommandPayload
*psPayload);
```

Description This function sends a Step Hue command to instruct a device to increase or decrease its 'current hue' attribute by a specified 'step' value in a continuous manner within a given time. The step size, direction and transition time are specified in the payload of the command (see Section 31.7.2).

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'hue and saturation' mode is selected by setting the 'colour mode' attribute to 0x00, if required. It can then move the 'current hue' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'current hue' attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN

- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandMoveToSaturationCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveToSaturationCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ColourControl_MoveToSaturationCommandPayload
*psPayload);
```

Description This function sends a Move to Saturation command to instruct a device to move its 'current saturation' attribute to a target saturation value in a continuous manner within a given time. The saturation value and transition time are specified in the payload of the command (see Section 31.7.2).

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'hue and saturation' mode is selected by setting the 'colour mode' attribute to 0x00, if required. It can then move the 'current saturation' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'current saturation' attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandMoveSaturationCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveSaturationCommandSend(
uint8 u8SourceEndPointId,
uint8 u8DestinationEndPointId,
tsZCL_Address *psDestinationAddress,
uint8 *pu8TransactionSequenceNumber,
tsCLD_ColourControl_MoveSaturationCommandPayload
*psPayload);
```

Description This function sends a Move Saturation command to instruct a device to move its 'current saturation' attribute value in a given direction at a specified rate for an indefinite time. The direction and rate are specified in the payload of the command (see Section 31.7.2).

The command can request that the saturation is moved up or down, or that existing movement is stopped. Once started, the movement will continue until it is stopped. If the current saturation reaches its minimum or maximum value, the movement will automatically stop.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'hue and saturation' mode is selected by setting the 'colour mode' attribute to 0x00, if required. It can then move the 'current saturation' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'current saturation' attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandStepSaturationCommandSend

```
teZCL_Status eCLD_ColourControlCommandStepSaturationCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ColourControl_StepSaturationCommandPayload
*psPayload);
```

Description This function sends a Step Saturation command to instruct a device to increase or decrease its 'current saturation' attribute by a specified 'step' value in a continuous manner within a given time. The step size, direction and transition time are specified in the payload of the command (see Section 31.7.2).

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'hue and saturation' mode is selected by setting the 'colour mode' attribute to 0x00, if required. It can then move the 'current saturation' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'current saturation' attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandMoveToHueAndSaturationCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveToColourCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ColourControl_MoveToColourCommandPayload
    *psPayload);
```

Description This function sends a Move to Hue and Saturation command to instruct a device to move its 'current hue' and 'current saturation' attributes to target values in a continuous manner within a given time. The hue value, saturation value and transition time are specified in the payload of the command (see Section 31.7.2).

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'hue and saturation' mode is selected by setting the 'colour mode' attribute to 0x00, if required. It can then move the 'current hue' and 'current saturation' values as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing

an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'current hue' and 'current saturation' attributes are enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandMoveToColourCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveToColourCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_MoveToColourCommandPayload
    *psPayload);
```

Description This function sends a Move to Colour command to instruct a device to move its 'current x' and 'current y' attributes to target values in a continuous manner within a given time (where x and y are the chromaticities from the CIE xyY colour space). The x-value, y-value and transition time are specified in the payload of the command (see Section 31.7.2).

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'chromaticities x and y' mode is selected by setting the 'colour mode' attribute to 0x01, if required. It can then move the 'current x' and 'current y' values as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'current x' and 'current y' attributes are enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandMoveColourCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveColourCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ColourControl_MoveColourCommandPayload
*psPayload);
```

Description This function sends a Move Colour command to instruct a device to move its 'current x' and 'current y' attribute values at a specified rate for each attribute for an indefinite time (where x and y are the chromaticities from the CIE xyY colour space). The rates are specified in the payload of the command (see Section 31.7.2 and each rate can be positive (increase) or negative (decrease).

Once started, the movement will continue until it is stopped. The movement can be stopped by calling this function with both rates set to zero. The movement will be automatically stopped when either of the attributes reaches its minimum of maximum value.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'chromaticities x and y' mode is selected by setting the 'colour mode' attribute to 0x01, if required. It can then move the 'current x' and 'current y' values as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'current x' and 'current y' values attributes are enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandStepColourCommandSend

```
teZCL_Status eCLD_ColourControlCommandStepColourCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ColourControl_StepColourCommandPayload
*psPayload);
```

Description This function sends a Step Colour command to instruct a device to change its 'current x' and 'current y' attribute values by a specified 'step' value for each attribute in a continuous manner within a given time (where x and y are the chromaticities from the CIE xyY colour space). The step sizes and transition time are specified in the payload of the command (see Section 31.7.2), and each step size can be positive (increase) or negative (decrease).

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'chromaticities x and y' mode is selected by setting the 'colour mode' attribute to 0x01, if required. It can then move the 'current x' and 'current y' values as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'current x' and 'current y' values attributes are enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL

- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandEnhancedMoveToHueCommandSend

```
teZCL_Status eCLD_ColourControlCommandEnhancedMoveToHueCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ColourControl_EnhancedMoveToHueCommandPayload
*psPayload);
```

Description This function sends an Enhanced Move to Hue command to instruct a device to move its 'enhanced current hue' attribute to a target hue value in a continuous manner within a given time. The enhanced hue value, direction and transition time are specified in the payload of the command (see Section 31.7.2). The 'current hue' attribute is also moved to a value based on the target 'enhanced current hue' value.

Since the possible hues are represented on a closed boundary, the target hue can be reached by moving the attribute value in either direction, up or down (the attribute value wraps around). Options are also provided for 'shortest route' and 'longest route' around the boundary.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'hue and saturation' mode is selected by setting the 'colour mode' attribute to 0x00 and that 'enhanced hue and saturation' mode is selected by setting the 'enhanced colour mode' attribute to 0x03, if required. It can then move the 'enhanced current hue' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'enhanced current hue' attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandEnhancedMoveHueCommandSend

```
teZCL_Status eCLD_ColourControlCommandEnhancedMoveHueCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ColourControl_EnhancedMoveHueCommandPayload
*psPayload);
```

Description This function sends an Enhanced Move Hue command to instruct a device to move its 'enhanced current hue' attribute value in a given direction at a specified rate for an indefinite time. The direction and rate are specified in the payload of the command (see Section 31.7.2). The 'current hue' attribute is also moved through values based on the 'enhanced current hue' value.

The command can request that the hue is moved up or down, or that existing movement is stopped. Since the possible hues are represented on a closed boundary, the movement is cyclic (the attribute value wraps around). Once started, the movement will continue until it is stopped.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'hue and saturation' mode is selected by setting the 'colour mode' attribute to 0x00 and that 'enhanced hue and saturation' mode is selected by setting the 'enhanced colour mode' attribute to 0x03, if required. It can then move the 'enhanced current hue' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'enhanced current hue' attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values

- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandEnhancedStepHueCommandSend

```
teZCL_Status eCLD_ColourControlCommandEnhancedStepHueCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ColourControl_EnhancedStepHueCommandPayload
*psPayload);
```

Description This function sends an Enhanced Step Hue command to instruct a device to increase or decrease its 'enhanced current hue' attribute by a specified 'step' value in a continuous manner within a given time. The step size, direction and transition time are specified in the payload of the command (see Section 31.7.2). The 'current hue' attribute is also moved through values based on the 'enhanced current hue' value.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'hue and saturation' mode is selected by setting the 'colour mode' attribute to 0x00 and that 'enhanced hue and saturation' mode is selected by setting the 'enhanced colour mode' attribute to 0x03, if required. It can then move the 'enhanced current hue' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'enhanced current hue' attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandEnhancedMoveToHueAndSaturationCommandSend

```
teZCL_Status eCLD_ColourControlCommandEnhancedMoveToHueAndSaturationCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_EnhancedMoveToHueAndSaturation
    CommandPayload *psPayload);
```

Description This function sends an Enhanced Move to Hue and Saturation command to instruct a device to move its 'enhanced current hue' and 'current saturation' attributes to target values in a continuous manner within a given time. The enhanced hue value, saturation value and transition time are specified in the payload of the command (see Section 31.7.2). The 'current hue' attribute is also moved to a value based on the target 'enhanced current hue' value.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'hue and saturation' mode is selected by setting the 'colour mode' attribute to 0x00 and that 'enhanced hue and saturation' mode is selected by setting the 'enhanced colour mode' attribute to 0x03, if required. It can then move the 'enhanced current hue' and 'current saturation' values as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'enhanced current hue' and 'current saturation' attributes are enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandColourLoopSetCommandSend

```
teZCL_Status eCLD_ColourControlCommandColourLoopSetCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_ColourLoopSetCommandPayload
    *psPayload);
```

Description This function sends a Colour Loop Set command to instruct a device to configure the movement of the 'enhanced current hue' attribute value around the colour loop corresponding to the CIE colour 'triangle'. The configured movement can be started in either direction and for a specific duration. The start hue, direction and duration are specified in the payload of the command (see Section 31.7.2). The 'current hue' attribute is also moved through values based on the 'enhanced current hue' value.

The function can also be used to stop existing movement around the colour loop.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'hue and saturation' mode is selected by setting the 'colour mode' attribute to 0x00 and that 'enhanced hue and saturation' mode is selected by setting the 'enhanced colour mode' attribute to 0x03, if required. It can then move the 'enhanced current hue' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'enhanced current hue' attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandStopMoveStepCommandSend

```
teZCL_Status eCLD_ColourControlCommandStopMoveStepCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ColourControl_StopMoveStepCommandPayload
*psPayload);
```

Description This function sends a Stop Move Step command to instruct a device to stop a 'Move to', 'Move' or 'Step' command that is currently in progress.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered, and stop the current action.

The 'current hue', 'enhanced current hue' and 'current saturation' attributes will subsequently keep the values they have when the current action is stopped.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'enhanced current hue' attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload* : Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandMoveToColourTemperatureCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveToColourTemperatureCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ColourControl_MoveToColourTemperatureCommandPayload
*psPayload);
```

Description This function sends a Move to Colour Temperature command to instruct a device to move its 'mired colour temperature' attribute to a target value in a continuous manner within a given time. The attribute value is a scaled reciprocal of colour temperature, as indicated in Section 31.5.4. The target attribute value, direction and transition time are specified in the payload of the command (see Section 31.7.2).

The movement through colour space will follow the 'Black Body Line'.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'colour temperature' mode is selected by setting the 'colour mode' attribute to 0x02, if required. It can then move the 'mired colour temperature' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'mired colour temperature' attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandMoveColourTemperatureCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveColourTemperatureCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ColourControl_MoveColourTemperatureCommandPayload
*psPayload);
```

Description This function sends a Move Colour Temperature command to instruct a device to move its 'mired colour temperature' attribute value in a given direction at a specified rate. The attribute value is a scaled reciprocal of colour temperature, as indicated in Section 31.5.4. The direction and rate are specified in the payload of the command (see Section 31.7.2). Maximum and minimum attribute values for the movement are also specified in the payload.

The command can request that the attribute value is moved up or down, or that existing movement is stopped. Once started, the movement will automatically stop when the attribute value reaches the specified maximum or minimum.

The movement through colour space will follow the 'Black Body Line'.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'colour temperature' mode is selected by setting the 'colour mode' attribute to 0x02, if required. It can then move the 'mired colour temperature' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'mired colour temperature' attribute is enabled in the Colour Control cluster, as well as the 'mired colour temperature maximum' and 'mired colour temperature minimum' attributes.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControlCommandStepColourTemperatureCommandSend

```
teZCL_Status eCLD_ColourControlCommandStepColourTemperatureCommandSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ColourControl_StepColourTemperatureCommandPayload
*psPayload);
```

Description This function sends a Step Colour Temperature command to instruct a device to increase or decrease its 'mired colour temperature' attribute by a specified 'step' value in a continuous manner within a given time. The attribute value is a scaled reciprocal of colour temperature, as indicated in Section 31.5.4. The step size, direction and transition time are specified in the payload of the command (see Section 31.7.2). Maximum and minimum attribute values for the movement are also specified in the payload.

The command can request that the attribute value is moved up or down. If this value reaches the specified maximum or minimum before the required change has been achieved, the movement will automatically stop.

The movement through colour space will follow the 'Black Body Line'.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'colour temperature' mode is selected by setting the 'colour mode' attribute to 0x02, if required. It can then move the 'mired colour temperature' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the 'mired colour temperature' attribute is enabled in the Colour Control cluster, as well as the 'mired colour temperature maximum' and 'mired colour temperature minimum' attributes.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- **psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_ColourControl_GetRGB

```
teZCL_Status eCLD_ColourControl_GetRGB(
    uint8 u8SourceEndPointId,
    uint8 *pu8Red,
    uint8 *pu8Green,
    uint8 *pu8Blue);
```

Description This function obtains the current colour of the device on the specified (local) endpoint in terms of the Red (R), Green (G) and Blue (B) components.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which the device resides
- *pu8Red*: Pointer to a location to receive the red value, in the range 0-255
- *pu8Green*: Pointer to a location to receive the green value, in the range 0-255

- *pu8Blue*: Pointer to a location to receive the blue value, in the range 0-255

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

Parent topic:*Colour Control Cluster*

Structures

Custom Data Structure The Colour Control cluster requires extra storage space to be allocated for use by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    teCLD_ColourControl_ColourMode      eColourMode;
    uint16                              u16CurrentHue;
    tsCLD_ColourControl_Transition      sTransition;
    /* Matrices for XYZ <> RGB conversions */
    float                               afXYZ2RGB[3][3];
    float                               afRGB2XYZ[3][3];
    tsZCL_ReceiveEventAddress           sReceiveEventAddress;
    tsZCL_CallBackEvent                 sCustomCallBackEvent;
    tsCLD_ColourControlCallBackMessage sCallBackMessage;
} tsCLD_ColourControlCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Custom Command Payloads The following structures contain the payloads for the Colour Control cluster custom commands.

Move to Hue Command Payload

```
typedef struct
{
    uint8                           u8Hue;
    teCLD_ColourControl_Direction   eDirection;
    uint16                          u16TransitionTime;
} tsCLD_ColourControl_MoveToHueCommandPayload;
```

where:

- u8Hue is the target hue value.
- eDirection indicates the direction/path of the change in hue:

eDirection	Direction/Path
0x00	Shortest path
0x01	Longest path
0x02	Up
0x03	Down
0x04 – 0xFF	Reserved

- u16TransitionTime is the time period, in tenths of a second, over which the change in hue should be implemented.

Move Hue Command Payload

```
typedef struct
{
    teCLD_ColourControl_MoveMode     eMode;
    uint8                            u8Rate;
    zbmap8                           u8OptionsMask;
    zbmap8                           u8OptionsOverride;
} tsCLD_ColourControl_MoveHueCommandPayload;
```

where:

- eMode indicates the required action and/or direction of the change in hue:

eMode	Action/Direction
0x00	Stop existing movement in hue
0x01	Start increasing hue
0x02	Reserved
0x03	Start decreasing hue
0x04 – 0xFF	Reserved

- u8Rate is the required rate of movement in hue steps per second (a step is one unit of hue for the device).
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Step Hue Command Payload

```
typedef struct
{
    teCLD_ColourControl_StepMode     eMode;
    uint8                            u8StepSize;
    uint8                            u8TransitionTime;
    zbmap8                           u8OptionsMask;
    zbmap8                           u8OptionsOverride;
} tsCLD_ColourControl_StepHueCommandPayload;
```

where:

- eMode indicates the required direction of the change in hue:

eMode	Action/Direction
0x00	Reserved
0x01	Increase hue
0x02	Reserved
0x03	Decrease hue
0x04 – 0xFF	Reserved

- u8StepSize is the amount by which the hue is to be changed (increased or decreased), in units of hue for the device.
- u8TransitionTime is the time period, in tenths of a second, over which the change in hue should be implemented.
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Move To Saturation Command Payload

```
typedef struct
{
    uint8                    u8Saturation;
    uint16                   u16TransitionTime;
    zbmap8                   u8OptionsMask;
    zbmap8                   u8OptionsOverride;
} tsCLD_ColourControl_MoveToSaturationCommandPayload;
```

where:

- u8Saturation is the target saturation value.
- u16TransitionTime is the time period, in tenths of a second, over which the change in saturation should be implemented.
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Move Saturation Command Payload

```
typedef struct
{
    teCLD_ColourControl_MoveMode    eMode;
    uint8                    u8Rate;
    zbmap8                   u8OptionsMask;
    zbmap8                   u8OptionsOverride;
} tsCLD_ColourControl_MoveSaturationCommandPayload;
```

where:

- eMode indicates the required action and/or direction of the change in saturation:

eMode	Action/Direction
0x00	Stop existing movement in hue
0x01	Start increasing saturation
0x02	Reserved
0x03	Start decreasing saturation
0x04 – 0xFF	Reserved

- u8Rate is the required rate of movement in saturation steps per second (a step is one unit of saturation for the device).
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Step Saturation Command Payload

```
typedef struct
{
    teCLD_ColourControl_StepMode     eMode;
    uint8                            u8StepSize;
    uint8                            u8TransitionTime;
    zbmap8                             u8OptionsMask;
    zbmap8                             u8OptionsOverride;
} tsCLD_ColourControl_StepSaturationCommandPayload;
```

where:

- eMode indicates the required direction of the change in saturation:

eMode	Action/Direction
0x00	Reserved
0x01	Increase saturation
0x02	Reserved
0x03	Decrease saturation
0x04 – 0xFF	Reserved

- u8StepSize is the amount by which the saturation is to be changed (increased or decreased), in units of saturation for the device.
- u8TransitionTime is the time period, in tenths of a second, over which the change in hue should be implemented.
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Move To Hue And Saturation Command Payload

```
typedef struct
{
    uint8                            u8Hue;
    uint8                            u8Saturation;
    uint16                            u16TransitionTime;
```

(continues on next page)

(continued from previous page)

```
    zbmap8                    u8OptionsMask;
    zbmap8                    u8OptionsOverride;
} tsCLD_ColourControl_MoveToHueAndSaturationCommandPayload;
```

where:

- u8Hue is the target hue value.
- u8Saturation is the target saturation value.
- 16TransitionTime is the time period, in tenths of a second, over which the change in hue and saturation should be implemented.
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Move To Colour Command Payload

```
typedef struct
{
    uint16                    u16ColourX;
    uint16                    u16ColourY;
    uint16                    u16TransitionTime;
    zbmap8                      u8OptionsMask;
    zbmap8                      u8OptionsOverride;
} tsCLD_ColourControl_MoveToColourCommandPayload;
```

where:

- u16ColourX is the target x-chromaticity in the CIE xyY colour space
- u16ColourY is the target y-chromaticity in the CIE xyY colour space
- u16TransitionTime is the time period, in tenths of a second, over which the colour change should be implemented.
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Move Colour Command Payload

```
typedef struct
{
    int16                     i16RateX;
    int16                     i16RateY;
    zbmap8                      u8OptionsMask;
    zbmap8                      u8OptionsOverride;
} tsCLD_ColourControl_MoveColourCommandPayload;
```

where:

- i16RateX is the required rate of movement of x-chromaticity in the CIE xyY colour space, in steps per second (a step is one unit of x-chromaticity for the device).

- i16RateY is the required rate of movement of y-chromaticity in the CIE xyY colour space, in steps per second (a step is one unit of y-chromaticity for the device).
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Step Colour Command Payload

```
typedef struct
{
    int16                    i16StepX;
    int16                    i16StepY;
    uint16                    u16TransitionTime;
    zbmap8                      u8OptionsMask;
    zbmap8                      u8OptionsOverride;
} tsCLD_ColourControl_StepColourCommandPayload;
```

where:

- i16StepX is the amount by which the x-chromaticity in the CIE xyY colour space is to be changed (increased or decreased), in units of x-chromaticity for the device.
- i16StepY is the amount by which the y-chromaticity in the CIE xyY colour space is to be changed (increased or decreased), in units of y-chromaticity for the device.
- u16TransitionTime is the time period, in tenths of a second, over which the colour change should be implemented.
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Move To Colour Temperature Command Payload

```
typedef struct
{
    uint16                    u16ColourTemperatureMired;
    uint16                    u16TransitionTime;
    zbmap8                      u8OptionsMask;
    zbmap8                      u8OptionsOverride;
} tsCLD_ColourControl_MoveToColourTemperatureCommandPayload;
```

where:

- u16ColourTemperature``Mired is the target value of the mired colour temperature attribute u16ColourTemperature``Mired (this value is a scaled reciprocal of colour temperature - for details, refer to the attribute description in Section 31.2).
- u16TransitionTime is the time period, in tenths of a second, over which the change in colour temperature should be implemented.
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Move Colour Temperature Command Payload

```
typedef struct
{
    teCLD_ColourControl_MoveMode      eMode;
    uint16                            u16Rate;
    uint16                            u16ColourTemperatureMiredMin;
    uint16                            u16ColourTemperatureMiredMax;
    zbmap8                            u8OptionsMask;
    zbmap8                            u8OptionsOverride;
} tsCLD_ColourControl_MoveColourTemperatureCommandPayload;
```

where:

- eMode indicates the required action and/or direction of the change in the mired colour temperature attribute value:

eMode	Action/Direction
0x00	Stop existing movement in colour temperature
0x01	Start increasing mired colour temperature attribute value
0x02	Reserved
0x03	Start decreasing mired colour temperature attribute value
0x04 – 0xFF	Reserved

- u16Rate is the required rate of movement in mired colour temperature steps per second (a step is one unit of the mired colour temperature attribute).
- u16ColourTemperature``Mired``Min is the lower limit for the mired colour temperature attribute during the operation resulting from this command.
- u16ColourTemperature``Mired``Max is the upper limit for the mired colour temperature attribute during the operation resulting from this command.
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Step Colour Temperature Command Payload

```
typedef struct
{
    teCLD_ColourControl_StepMode      eMode;
    uint16                            u16StepSize;
    uint16                            u16TransitionTime;
    uint16                            u16ColourTemperatureMiredMin;
    uint16                            u16ColourTemperatureMiredMax;
    zbmap8                            u8OptionsMask;
    zbmap8                            u8OptionsOverride;
} tsCLD_ColourControl_StepColourTemperatureCommandPayload;
```

where:

- eMode indicates the required direction of the change in the mired colour temperature attribute value:

eMode	Action/Direction
0x00	Reserved
0x01	Increase mired colour temperature attribute value
0x02	Reserved
0x03	Decrease mired colour temperature attribute value
0x04 – 0xFF	Reserved

- u16StepSize is the amount by which the mired colour temperature attribute is to be changed (increased or decreased).
- u16TransitionTime is the time period, in tenths of a second, over which the change in the mired colour temperature attribute should be implemented.
- u16ColourTemperature``Mired``Min is the lower limit for the mired colour temperature attribute during the operation resulting from this command.
- u16ColourTemperature``Mired``Max is the upper limit for the mired colour temperature attribute during the operation resulting from this command.
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Enhanced Move To Hue Command Payload

```
typedef struct
{
    uint16                          u16EnhancedHue;
    teCLD_ColourControl_Direction       eDirection;
    uint16                          u16TransitionTime;
    zbmap8                            u8OptionsMask;
    zbmap8                            u8OptionsOverride;
} tsCLD_ColourControl_EnhancedMoveToHueCommandPayload;
```

where:

- u16EnhancedHue is the target 'enhanced' hue value in terms of a step around the CIE colour 'triangle' - for the format, refer to the description of the attribute u16EnhancedCurrentHue in Section 31.2.
- eDirection indicates the direction/path of the change in hue:

eDirection	Direction/Path
0x00	Shortest path
0x01	Longest path
0x02	Up
0x03	Down
0x04 – 0xFF	Reserved

- u16TransitionTime is the time period, in tenths of a second, over which the change in hue should be implemented.
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Enhanced Move Hue Command Payload

```
typedef struct
{
    teCLD_ColourControl_MoveMode        eMode;
    uint16                              u16Rate;
    zbmap8                              u8OptionsMask;
    zbmap8                              u8OptionsOverride;
} tsCLD_ColourControl_EnhancedMoveHueCommandPayload;
```

where:

- eMode indicates the required action and/or direction of the change in hue:

eMode	Action/Direction
0x00	Stop existing movement in hue
0x01	Start increase in hue
0x02	Reserved
0x03	Start decrease in hue
0x04 – 0xFF	Reserved

- u16Rate is the required rate of movement in 'enhanced' hue steps per second (a step is one unit of hue for the device).
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Enhanced Step Hue Command Payload

```
typedef struct
{
    teCLD_ColourControl_StepMode        eMode;
    uint16                              u16StepSize;
    uint16                              u16TransitionTime;
    zbmap8                              u8OptionsMask;
    zbmap8                              u8OptionsOverride;
} tsCLD_ColourControl_EnhancedStepHueCommandPayload;
```

where:

- eMode indicates the required direction of the change in hue:

eMode	Action/Direction
0x00	Reserved
0x01	Increase in hue
0x02	Reserved
0x03	Decrease in hue
0x04 – 0xFF	Reserved

- u16StepSize is the amount by which the 'enhanced' hue is to be changed (increased or decreased) - for the format, refer to the description of the attribute u16EnhancedCurrentHue in Section 31.2.
- u8TransitionTime is the time period, in tenths of a second, over which the change in hue should be implemented.

- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Enhanced Move To Hue And Saturation Command Payload

```
typedef struct
{
    uint16                          u16EnhancedHue;
    uint8                           u8Saturation;
    uint16                          u16TransitionTime;
    zbmap8                          u8OptionsMask;
    zbmap8                          u8OptionsOverride;
} tsCLD_ColourControl_EnhancedMoveToHueAndSaturationCommandPayload;
```

where:

- u16EnhancedHue is the target 'enhanced' hue value in terms of a step around the CIE colour 'triangle' - for the format, refer to the description of the attribute u16EnhancedCurrentHue in Section 31.2.
- u8Saturation is the target saturation value.
- 16TransitionTime is the time period, in tenths of a second, over which the change in hue and saturation should be implemented.
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Colour Loop Set Command Payload

```
typedef struct
{
    uint8                           u8UpdateFlags;
    teCLD_ColourControl_LoopAction      eAction;
    teCLD_ColourControl_LoopDirection   eDirection;
    uint16                          u16Time;
    uint16                          u16StartHue;
    zbmap8                          u8OptionsMask;
    zbmap8                          u8OptionsOverride;
} tsCLD_ColourControl_ColourLoopSetCommandPayload;
```

where:

- u8UpdateFlags is a bitmap indicating which of the other fields of the structure must be set (a bit must be set to '1' to enable the corresponding field, and '0' otherwise):

Bits	Field
0	‘‘‘
eAction	

```
|
|1|```
eDirection
```

| |2|"' u16Time

```
|
|3|```
u16StartHue
```

| |4–7|Reserved|

- eAction indicates the colour loop action to be taken (if enabled through u8UpdateFlags), as one of:

Enumeration	Value	Action
E_CLD_COLOURCONTROL_COLOURLOOP_ACTION_		

DEACTIVATE

|0x00|Deactivate colour loop| |E_CLD_COLOURCONTROL_COLOURLOOP_ACTION_

ACTIVATE_FROM_START

|0x01|Activate colour loop from specified start (enhanced) hue value| |E_CLD_COLOURCONTROL_COLOURLOOP_ACTION_

ACTIVATE_FROM_CURRENT

|0x02|Activate colour from current (enhanced) hue value|

- eDirection indicates the direction to be taken around the colour loop (if enabled through u8UpdateFlags) in terms of the direction of change of u16EnhancedCurrentHue:

Enumeration	Value	Direction
E_CLD_COLOURCONTROL_COLOURLOOP_		

DIRECTION_DECREMENT

|0x00|Decrement current (enhanced) hue value| |E_CLD_COLOURCONTROL_COLOURLOOP_

DIRECTION_INCREMENT

|0x01|Increment current (enhanced) hue value|

- u16Time is the period, in seconds, of a full colour loop - that is, the time to cycle all possible values of u16EnhancedCurrentHue.
- u16StartHue is the value of u16EnhancedCurrentHue at which the colour loop is to be started (if enabled through u8UpdateFlags).
- OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Stop Move Step Command Payload

```
typedef struct
{
    zbmap8              u8OptionsMask;
    zbmap8              u8OptionsOverride;
} tsCLD_ColourControl_StopMoveStepCommandPayload;
```

where OptionsMask and OptionsOverride must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the u8Options attribute. Each bit of the u8Options attribute is carried across to the temporary Options bitmap unless the corresponding bit of OptionsMask is set (to 1). In this case, the corresponding bit of OptionsOverride is used in the temporary Options bitmap instead.

Parent topic: Structures

Parent topic:*Colour Control Cluster*

Enumerations

teCLD_ColourControl_ClusterID The following structure contains the enumerations used to identify the attributes of the Colour Control cluster.

```
typedef enum
{
    E_CLD_COLOURCONTROL_ATTR_CURRENT_HUE              = 0x0000,
    E_CLD_COLOURCONTROL_ATTR_CURRENT_SATURATION,
    E_CLD_COLOURCONTROL_ATTR_REMAINING_TIME,
    E_CLD_COLOURCONTROL_ATTR_CURRENT_X,
    E_CLD_COLOURCONTROL_ATTR_CURRENT_Y,
    E_CLD_COLOURCONTROL_ATTR_DRIFT_COMPENSATION,
    E_CLD_COLOURCONTROL_ATTR_COMPENSATION_TEXT,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIRED,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_MODE,
    E_CLD_COLOURCONTROL_ATTR_OPTIONS        = 0x000F,
    E_CLD_COLOURCONTROL_ATTR_NUMBER_OF_PRIMARIES     = 0x0010,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_1_X,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_1_Y,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_1_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_2_X            = 0x0015,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_2_Y,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_2_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_3_X            = 0x0019,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_3_Y,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_3_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_4_X            = 0x0020,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_4_Y,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_4_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_5_X            = 0x0024,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_5_Y,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_5_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_6_X            = 0x0028,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_6_Y,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_6_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_WHITE_POINT_X          = 0x0030,
    E_CLD_COLOURCONTROL_ATTR_WHITE_POINT_Y,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_X,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_Y,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_X         = 0x0036,
```

(continues on next page)

(continued from previous page)

```
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_Y,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_X          = 0x003A,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_Y,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_ENHANCED_CURRENT_HUE       = 0x4000,
    E_CLD_COLOURCONTROL_ATTR_ENHANCED_COLOUR_MODE,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_ACTIVE,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_DIRECTION,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_TIME,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_START_ENHANCED_HUE,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_STORED_ENHANCED_HUE,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_CAPABILITIES        = 0x400a,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIRED_PHY_MIN,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIRED_PHY_MAX,
    E_CLD_COLOURCONTROL_ATTR_COUPLE_COLOUR_TEMPERATURE_TO_LEVEL_MIN_
↪MIRED,
    E_CLD_COLOURCONTROL_ATTR_STARTUP_COLOUR_TEMPERATURE_MIRED = 0x4010,
} teCLD_ColourControl_ClusterID;
```

Parent topic: Enumerations

Parent topic:*Colour Control Cluster*

Compile-time Options To enable the Colour Control cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_COLOUR_CONTROL
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define COLOUR_CONTROL_CLIENT
#define COLOUR_CONTROL_SERVER
```

The Colour Cluster cluster attributes reside on the server only. Therefore, attributes should not be enabled in the **zcl_options.h** file for the cluster client.

Optional Attributes The optional attributes of the Colour Control cluster are enabled/disabled by defining the following in the **zcl_options.h** file:

- For optional attributes from the 'Colour Information' attribute set:
  - CLD_COLOURCONTROL_ATTR_REMAINING_TIME
  - CLD_COLOURCONTROL_ATTR_DRIFT_COMPENSATION
  - CLD_COLOURCONTROL_ATTR_COMPENSATION_TEXT
  - CLD_COLOURCONTROL_ATTR_COLOUR_MODE
    - Certain attributes from this attribute set are enabled through a '*Colour*Capabilities' Definition (see below) - these are u8CurrentHue, u8CurrentSaturation and u16ColourTemperatureMired.
- For optional attributes from the 'Defined Primaries Information' and 'Additional Defined Primaries Information' attribute sets, the macro
  - CLD_COLOURCONTROL_ATTR_NUMBER_OF_PRIMARIES

    * is used to define the required number of colour primaries, N, in the range 1 to 6 (0xFF can also be specified if the number of primaries is not known). This macro is used to automatically enable the required attributes from these attribute sets - for example, if N is set to 4 then the following attributes are enabled:
    * u16Primary1X, u16Primary1Y, u8Primary1Intensity, u16Primary2X, u16Primary2Y, u8Primary2Intensity, u16Primary3X, u16Primary3Y, u8Primary3Intensity, u16Primary4X, u16Primary4Y, u8Primary4Intensity.

- For optional attributes from the 'Defined Colour Points Settings' attribute set:
  - CLD_COLOURCONTROL_ATTR_WHITE_POINT_X
  - CLD_COLOURCONTROL_ATTR_WHITE_POINT_Y
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_X
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_Y
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_INTENSITY
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_X
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_Y
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_INTENSITY
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_X
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_Y
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_INTENSITY
- For optional attributes from the 'Enhanced Colour Mode' attributes, the following must be defined:
  - CLD_COLOURCONTROL_ATTR_ENHANCED_COLOUR_MODE
  - CLD_COLOURCONTROL_ATTR_COLOUR_CAPABILITIES
    * The required 'Enhanced Colour Mode' attributes for a device must then be enabled through a *'Colour Capabilities' Definition* (see below).

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_COLOURCONTROL_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_COLOURCONTROL_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

'Colour Capabilities' Definition If required, certain 'Colour Information' attributes and all 'Enhanced Colour Mode' attributes must be enabled through a 'Colour Capabilities' definition. Attributes are enabled as a group according to the required capability/functionality. The capabilities are detailed in the table below, with their corresponding attributes and macros.

The above macros automatically invoke the macros for the individual attributes in the capability group. For example, E_CLD_COLOURCONTROL_ATTR_CURRENT_HUE is invoked for the attribute u8CurrentHue.

The enabled Colour Capabilities are reflected in the 'Enhanced Colour Mode' attribute (bitmap) u16ColourCapabilities.

Example Colour Capabilities definitions are provided below for different devices.

ZLO Extended Colour Light:

```
#define CLD_COLOURCONTROL_COLOUR_CAPABILITIES
        (COLOUR_CAPABILITY_HUE_SATURATION_SUPPORTED | \\
       COLOUR_CAPABILITY_ENHANCE_HUE_SUPPORTED  | \\
       COLOUR_CAPABILITY_COLOUR_LOOP_SUPPORTED | \\
       COLOUR_CAPABILITY_XY_SUPPORTED | \\
       COLOUR_CAPABILITY_COLOUR_TEMPERATURE_SUPPORTED)
```

ZLO Colour Light:

```
#define CLD_COLOURCONTROL_COLOUR_CAPABILITIES
           (COLOUR_CAPABILITY_HUE_SATURATION_SUPPORTED | \\
           COLOUR_CAPABILITY_ENHANCE_HUE_SUPPORTED    | \\
           COLOUR_CAPABILITY_COLOUR_LOOP_SUPPORTED    | \\
           COLOUR_CAPABILITY_XY_SUPPORTED)
```

ZLO Colour Temperature Light:

```
#define CLD_COLOURCONTROL_COLOUR_CAPABILITIES
      (COLOUR_CAPABILITY_COLOUR_TEMPERATURE_SUPPORTED)
```

Parent topic:*Colour Control Cluster*

Ballast Configuration Cluster This chapter describes the Ballast Configuration cluster, which is concerned with a configuring a lighting ballast that restricts the amount of light emitted by a set of lamps connected to the ballast.

The Ballast Configuration cluster has a Cluster ID of 0x0301.

- **Overview**
- **Cluster structure and attributes**
- **Functions**
- **Enumerations**
- **Compile-time options**

Part VI: HVAC Clusters This part comprises three chapters:

- *Chapter 33* details the **Thermostat** cluster
- *Chapter 34* details the **Fan Control** cluster
- *Chapter 35* details the **Thermostat UI Configuration** cluster

Thermostat Cluster This chapter outlines the Thermostat cluster, which provides an interface for configuring and controlling the functionality of a thermostat.

The Thermostat cluster has a Cluster ID of 0x0201.

Overview The Thermostat cluster is required in ZigBee devices as indicated in the table below.

	Server-side	Client-side
Mandatory in...	Thermostat	
Optional in...		Remote Control

The Thermostat cluster is enabled by defining CLD_THERMOSTAT in the **zcl_options.h** file.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Thermostat cluster are fully detailed in Section33.10.

The information that can potentially be stored in this cluster is organised into the following attribute sets:

- Thermostat Information
- Thermostat Settings

The attributes are listed and described next, in Section 33.2.

Parent topic:*Thermostat Cluster*

Thermostat Cluster structure and attributes The Thermostat cluster is contained in the following tsCLD_Thermostat structure:

```
typedef struct
{
#ifdef THERMOSTAT_SERVER
    zint16              i16LocalTemperature;
#ifdef CLD_THERMOSTAT_ATTR_OUTDOOR_TEMPERATURE
    zint16              i16OutdoorTemperature;
#endif
#ifdef CLD_THERMOSTAT_ATTR_OCCUPANCY
    zbmap8                u8Occupancy;
#endif
#ifdef CLD_THERMOSTAT_ATTR_ABS_MIN_HEAT_SETPOINT_LIMIT
    zint16              i16AbsMinHeatSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_ABS_MAX_HEAT_SETPOINT_LIMIT
    zint16              i16AbsMaxHeatSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_ABS_MIN_COOL_SETPOINT_LIMIT
    zint16              i16AbsMinCoolSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_ABS_MAX_COOL_SETPOINT_LIMIT
    zint16              i16AbsMaxCoolSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_PI_COOLING_DEMAND
    zuint8              u8PICoolingDemand;
#endif
#ifdef CLD_THERMOSTAT_ATTR_PI_HEATING_DEMAND
    zuint8              u8PIHeatingDemand;
#endif
/* Thermostat settings attribute set attribute IDs */
#ifdef CLD_THERMOSTAT_ATTR_LOCAL_TEMPERATURE_CALIBRATION
```

(continues on next page)

(continued from previous page)

```
    zint8                   i8LocalTemperatureCalibration;
#endif
    zint16                  i16OccupiedCoolingSetpoint;
    zint16                  i16OccupiedHeatingSetpoint;
#ifdef CLD_THERMOSTAT_ATTR_UNOCCUPIED_COOLING_SETPOINT
    zint16                  i16UnoccupiedCoolingSetpoint;
#endif
#ifdef CLD_THERMOSTAT_ATTR_UNOCCUPIED_HEATING_SETPOINT
    zint16                  i16UnoccupiedHeatingSetpoint;
#endif
#ifdef CLD_THERMOSTAT_ATTR_MIN_HEAT_SETPOINT_LIMIT
    zint16                  i16MinHeatSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_MAX_HEAT_SETPOINT_LIMIT
    zint16                  i16MaxHeatSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_MIN_COOL_SETPOINT_LIMIT
    zint16                  i16MinCoolSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_MAX_COOL_SETPOINT_LIMIT
    zint16                  i16MaxCoolSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_MIN_SETPOINT_DEAD_BAND
    zint8                   i8MinSetpointDeadBand;
#endif
#ifdef CLD_THERMOSTAT_ATTR_REMOTE_SENSING
    zbmap8                  u8RemoteSensing;
#endif
    zenum8                  eControlSequenceOfOperation;
    zenum8                  eSystemMode;
#ifdef CLD_THERMOSTAT_ATTR_ALARM_MASK
    zbmap8                  u8AlarmMask;
#endif
#ifdef CLD_THERMOSTAT_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8                  u8AttributeReportingStatus;
#endif
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_Thermostat;
```

where:

'Thermostat Information' Attribute Set

- i16LocalTemperature is a mandatory attribute representing the measured temperature in degrees Celsius, as follows:
  - i16LocalTemperature = 100 x temperature in degrees Celsius
  - The possible values are used as follows:
  - 0x0000 to 0x7FFF represent positive temperatures from 0°C to 327.67ºC
  - 0x8000 indicates that the temperature measurement is invalid
  - 0x8001 to 0x954C are unused values
  - 0x954D to 0xFFFF represent negative temperatures from -273.15°C to
  - -1°C (in two's complement form)
- i16OutdoorTemperature is an optional attribute representing the outside temperature in degrees Celsius. This temperature is represented as described above for i16LocalTemperature.

- u8Occupancy is an optional attribute indicating whether the heated/cooled space has been detected as occupied. Bit 0 is used as a flag as follows (all other bits are reserved):
  - 1 = occupied
  - 0 = not occupied
- i16AbsMinHeatSetpointLimit is an optional attribute specifying the absolute minimum possible temperature of the heating setpoint (as determined by the manufacturer). This temperature is represented as described above for i16LocalTemperature.
- i16AbsMaxHeatSetpointLimit is an optional attribute specifying the absolute maximum possible temperature of the heating setpoint (as determined by the manufacturer). This temperature is represented as described above for i16LocalTemperature.
- i16AbsMinCoolSetpointLimit is an optional attribute specifying the absolute minimum possible temperature of the cooling setpoint (as determined by the manufacturer). This temperature is represented as described above for i16LocalTemperature.
- i16AbsMaxCoolSetpointLimit is an optional attribute specifying the absolute maximum possible temperature of the cooling setpoint (as determined by the manufacturer). This temperature is represented as described above for i16LocalTemperature.

'Thermostat Settings' Attribute Set

- u8PICoolingDemand is an optional attribute indicating the level of cooling required by the PI (Proportional Integral) control loop, if any, used by the thermostat. It is a percentage value and takes the value 0 when the thermostat is 'off' or in 'heating' mode.
- u8PIHeatingDemand is an optional attribute indicating the level of heating required by the PI (Proportional Integral) control loop, if any, used by the thermostat. It is a percentage value and takes the value 0 when the thermostat is 'off' or in 'cooling' mode.
- i8LocalTemperatureCalibration is an optional attribute representing a temperature offset (in the range -2.5°C to 2.5°C) that can be added to or subtracted from the displayed temperature:
  - i8LocalTemperatureCalibration = 100 x offset in degrees Celsius
  - The possible values are used as follows:
  - 0x00 to 0x19 represent positive offsets from 0°C to 2.5°C
  - 0x20 to 0xE6 are unused values
  - 0xE7 to 0xFF represent negative offets from -2.5°C to -1°C (in two's complement form)
- i16OccupiedCoolingSetpoint is an optional attribute specifying the cooling setpoint (target temperature) when the cooling space is occupied. The value is calculated as described above for the i16LocalTemperature attribute and must take a value in the range defined by the attributes i16MinCoolSetpointLimit and i16MaxCoolSetpointLimit. If it is not known whether the space is occupied, this attribute will be used as the cooling setpoint (rather than i16UnoccupiedCoolingSetpoint).
- i16OccupiedHeatingSetpoint is an optional attribute specifying the heating setpoint (target temperature) when the heating space is occupied. The value is calculated as described above for the i16LocalTemperature attribute and must take a value in the range defined by the attributes i16MinHeatSetpointLimit and i16MaxHeatSetpointLimit. If it is not known whether the space is occupied, this attribute will be used as the heating setpoint (rather than i16UnoccupiedHeatingSetpoint).

Note: i16OccupiedCoolingSetpoint must always be greater in value than i16OccupiedHeatingSetpoint by an amount at least equal to the value of i8MinSetpointDeadBand (below). An attempt to violate this condition will result in a default response with the status INVALID_VALUE.

- i16UnoccupiedCoolingSetpoint is an optional attribute specifying the cooling setpoint (target temperature) when the cooling space is unoccupied. The value is calculated as described

above for the i16LocalTemperature attribute and must take a value in the range defined by the attributes i16AbsMinCoolSetpointLimit and i16MaxCoolSetpointLimit. If it is not known whether the space is occupied, this attribute will not be used (i16OccupiedCoolingSetpoint will be used instead).

- i16``UnoccupiedHeatingSetpoint is an optional attribute specifying the heating setpoint (target temperature) when the heating space is unoccupied. The value is calculated as described above for the i16LocalTemperature attribute and must take a value in the range defined by the attributes i16Min``HeatS``etpointLimit and i16MaxHeatSetpointLimit. If it is not known whether the space is occupied, this attribute will not be used (i16Occupied``Heati``ngSetpoint will be used instead).

Note: i16UnoccupiedCoolingSetpoint must always be greater in value than i16``Unoc``cupiedHeatingSetpoint by an amount at least equal to the value of i8MinSetpointDeadBand (below). An attempt to violate this condition will result in a default response with the status INVALID_VALUE.

- i16MinHeatSetpointLimit is an optional attribute specifying the minimum possible temperature of the heating setpoint. This temperature is represented as described above for i16LocalTemperature. The value set must be greater than or equal to the value of i16AbsMinHeatSetpointLimit, which is also the default value for this attribute.
- i16MaxHeatSetpointLimit is an optional attribute specifying the maximum possible temperature of the heating setpoint. This temperature is represented as described above for i16LocalTemperature. The value set must be less than or equal to the value of i16AbsMaxHeatSetpointLimit, which is also the default value for this attribute.
- i16MinCoolSetpointLimit is an optional attribute specifying the minimum possible temperature of the cooling setpoint. This temperature is represented as described above for i16LocalTemperature. The value set must be greater than or equal to the value of i16AbsMinCoolSetpointLimit, which is also the default value for this attribute.
- i16MaxCoolSetpointLimit is an optional attribute specifying the maximum possible temperature of the cooling setpoint. This temperature is represented as described above for i16LocalTemperature. The value set must be less than or equal to the value of i16AbsMaxCoolSetpointLimit, which is also the default value for this attribute.

Note: The above four 'Limit' attributes can be set in the compile-time options using macros, as described in Section 33.10.

- i8MinSetpointDeadBand is an optional attribute specifying the minimum difference between the heating setpoint and cooling setpoint, in steps of 0.1°C. The attribute can take a value in the range 0x0A to 0x19, representing 1°C to 2.5°C. All other values are unused.
- u8RemoteSensing is an optional attribute comprising an 8-bit bitmap which indicates whether remote (networked) or internal sensors are being used to measure/detect the local temperature, outside temperature and occupancy. The bitmap is detailed in the table below.

Bit	Description
0	**Local temperature**

1 - Remote sensor

0 - Internal sensor

| |1|**Outside temperature**

1 - Remote sensor

0 - Internal sensor

| |2|**Occupancy**

1 - Remote sensor

0 - Internal sensor

| |3-7|Reserved|

- eControlSequenceOfOperation is an optional attribute representing the operational capabilities/environment of the thermostat. The possible values are indicated in the table below:

Value	Capabilities	Notes (see eSystemMode)
0x00	Cooling only	Heat and Emergency Heating are not possible
0x01	Cooling with Reheat	Heat and Emergency Heating are not possible
0x02	Heating only	Cool and Pre-cooling are not possible
0x03	Heating with Reheat	Cool and Pre-cooling are not possible
0x04	Cooling and Heating	

4-pipes

|All modes are possible| |0x05|Cooling and Heating

4-pipes with Reheat

|All modes are possible| |0x06 – 0xFE|Reserved|-|

- eSystemMode is an optional attribute specifying the current operating mode of the thermostat. The possible modes/values are indicated in the table below:

Value	Description
0x00	Off
0x01	Auto
0x02	Reserved
0x03	Cool
0x04	Heat
0x05	Emergency Heating
0x06	Pre-cooling
0x07	Fan only
0x08 – 0xFE	Reserved

- u8AlarmMask is an optional attribute containing a 3-bit bitmap specifying which alarms are enabled from those listed in the table below (use of the Alarms cluster is also required):

Bit	Description
0	**Initialisation failure** (device failed to complete initialization at power-up)

1 - Alarm enabled

0 - Alarm disabled

| |1|**Hardware failure**

1 - Alarm enabled

0 - Alarm disabled

| |2|**Self-calibration failure**

1 - Alarm enabled

0 - Alarm disabled

| |3-7|Reserved|

Global Attributes

- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Thermostat Cluster*

Attributes for Default Reporting The following attributes of the Thermostat cluster can be selected for default reporting:

```
i16LocalTemperature
u8PICoolingDemand
```

- u8PIHeatingDemand

Attribute reporting (including default reporting) is described in *Appendix B*. Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic:*Thermostat Cluster*

Thermostat Operations The Thermostat cluster server is mandatory for some HVAC devices, such as the Thermostat device, while the cluster client can be used on a controlling device, such as the Remote Control device.

The sections below describe common operations using the Thermostat cluster.

Initialisation The function **eCLD_ThermostatCreateThermostat()** is used to create an instance of the Thermostat cluster. The function is generally called by the initialization function for the host device.

Parent topic: Thermostat Operations

Recording and Reporting the Local Temperature A record of the local temperature is kept in the mandatory attribute i16LocalTemperature on the cluster server - this attribute is fully detailed in Section 33.2. The value of this attribute can be updated by the server application using the function **eCLD_ThermostatSetAttribute()** - for example, as the result of a local temperature measurement.

The value of the attribute i16LocalTemperature can be regularly reported to a cluster client - for example, to allow the local temperature to be displayed to the user. This automated reporting can be configured and started on the server using the function **eCLD_ThermostatStartReportingLocalTemperature()**. Reports is sent regularly, but not periodically - maximum and minimum time-intervals between consecutive reports can be specified.

Parent topic: Thermostat Operations

Configuring Heating and Cooling Setpoints Functions are provided to update the following two optional attributes that are used to specify setpoints (target temperatures) for heating and cooling:

```
i16OccupiedHeatingSetpoint
i16OccupiedCoolingSetpoint
```

If both of these setpoints are used, the cooling setpoint value must be greater than the heating setpoint value. These attributes are fully detailed in Section 33.2.

These server attributes can be controlled remotely from a client using the function **eCLD_ThermostatCommandSetpointRaiseOrLowerSend()**, usually as the result of user input on a controlling device. This function is used on the client to send a SetpointRaiseOrLower command to the server to increase or decrease the value of one or both of these setpoint attributes by a specified amount. On receipt of this command, an E_CLD_THERMOSTAT_CMD_SETPOINT_RAISE_LOWER event is generated on the server to notify the server application.

The server application can modify the values of these attributes using the function **eCLD_ThermostatSetAttribute()**.

Note: These and other attributes of the Thermostat cluster can also be written and read using the general attribute access functions, as described in Section 2.3.

Parent topic: Thermostat Operations

Parent topic:*Thermostat Cluster*

Thermostat Events The Thermostat cluster has its own events that are handled through the callback mechanism outlined in *Chapter 3*. If a device uses the Thermostat cluster then Thermostat event handling must be included in the callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function (for example, through **eHA_RegisterThermostatEndPoint()** for a Thermostat device). The relevant callback function will then be invoked when a Thermostat event occurs.

For a Thermostat event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_ThermostatCallBackMessage structure:

```
typedef struct
{
    uint8                    u8CommandId;
    union
    {
      tsCLD_Thermostat_SetpointRaiseOrLowerPayload
                            *psSetpointRaiseOrLowerPayload;
    } uMessage;
} tsCLD_ThermostatCallBackMessage;
```

The u8CommandId field of the above structure specifies the type of command that has been received - only one command type is possible and is described below.

E_CLD_THERMOSTAT_CMD_SETPOINT_RAISE_LOWER In the tsCLD_ThermostatCallBackMessage structure, the u8CommandId is set to E_CLD_THERMOSTAT_CMD_SETPOINT_RAISE_LOWER on the Thermostat cluster server when a SetpointRaiseOrLower command has been received. On receipt of this command, the Thermostat command handler will be invoked.

Parent topic:*Thermostat Cluster*

Functions The following Thermostat cluster functions are provided:

- **Function Page**
- eCLD_ThermostatCreateThermostat 733
- eCLD_ThermostatSetAttribute 735
- eCLD_ThermostatStartReportingLocalTemperature 736
- eCLD_ThermostatCommandSetpointRaiseOrLowerSend 737

eCLD_ThermostatCreateThermostat

```
teZCL_Status eCLD_ThermostatCreateThermostat(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    sZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits,
    tsCLD_ThermostatCustomDataStructure
    psCustomDataStructure);
```

Description This function creates an instance of the Thermostat cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Thermostat cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device (e.g. the Thermostat device) will be used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be the first Thermostat cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Thermostat cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Thermostat cluster. This parameter can refer to a pre-filled structure called sCLD_``Thermostat which is provided in the **Thermostat.h** file.

- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_``Thermostat which defines the attributes of Thermostat cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_ThermostatSetAttribute

```
teZCL_Status eCLD_ThermostatSetAttribute(
    uint8 u8SourceEndPointId,
    uint8 u8AttributeId,
    int16 i16AttributeValue);
```

Description This function can be used on a Thermostat cluster server to update the Thermostat attributes - specifically to write a value to one of the following attributes:

```
i16LocalTemperature
i16OccupiedCoolingSetpoint
i16OccupiedHeatingSetpoint
```

The function first checks whether the value to be written falls within the valid range for the relevant attribute. If not, it returns with status E_ZCL_ERR_INVALID_VALUE. If the server attempts to write to an attribute other than those specified above, the function returns with status E_ZCL_DENY_ATTRIBUTE_ACCESS. If the cluster does not exist, it returns with status E_ZCL_ERR_CLUSTER_NOT_FOUND.

Parameters

- *u8SourceEndPointId* Number of the endpoint on which the Thermostat cluster resides
- *u8AttributeId* Identifier of attribute to be updated, one of:
- E_CLD_THERMOSTAT_ATTR_ID_LOCAL_TEMPERATURE
- E_CLD_THERMOSTAT_ATTR_ID_OCCUPIED_COOLING_SETPOINT
- E_CLD_THERMOSTAT_ATTR_ID_OCCUPIED_HEATING_SETPOINT
- *i16AttributeValue* Value to be written to attribute

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_INVALID_VALUE

- E_ZCL_DENY_ATTRIBUTE_ACCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND

Parent topic: Functions

eCLD_ThermostatStartReportingLocalTemperature

```
teZCL_Status eCLD_ThermostatStartReportingLocalTemperature(
    uint8 u8SourceEndPointId,
    uint8 u8DstEndPointId,
    uint64 u64DstAddr,
    uint16 u16MinReportInterval,
    uint16 u16MaxReportInterval,
    int16 i16ReportableChange);
```

Description This function can be used on a Thermostat cluster server to start automatic reporting of the measured local temperature to a cluster client. The change to be reported can be configured through this function. Reports is sent regularly (but not periodically), within the specified maximum and minimum time-intervals between consecutive reports.

Parameters

- *u8SourceEndPointId* Number of the local endpoint on which the Thermostat cluster server resides
- *u8DstEndPointId* Number of the endpoint to which reports are to be sent on the destination node
- *u64DstAddr* IEEE/MAC address of destination node
- *u16MinReportInterval* Minimum time-interval, in seconds, between reports
- *u16MaxReportInterval* Maximum time-interval, in seconds, between reports
- *i16ReportableChange* Specifies the change to be reported

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_CLUSTER_NOT_FOUND

Parent topic: Functions

eCLD_ThermostatCommandSetpointRaiseOrLowerSend

```
teZCL_Status eCLD_ThermostatCommandSetpointRaiseOrLowerSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_Thermostat_SetpointRaiseOrLowerPayload
*psPayload);
```

Description This function can be used on a Thermostat cluster client to send a 'Setpoint Raise Or Lower' command to the cluster server. This command is used to increase or decrease the heating setpoint and/or cooling setpoint by requesting a change to the values of the attribute i16OccupiedHeatingSetpoint and/or the attribute i16OccupiedCoolingSetpoint. The relevant setpoint(s) and the required temperature change are specified in the command payload structure tsCLD_Thermostat_SetpointRaiseOrLowerPayload (see Section 33.9.3).

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId* Number of the remote endpoint to which the request is sent
- *psDestinationAddress* Pointer to a structure containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to the command payload (see Section33.9.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Functions

Parent topic:*Thermostat Cluster*

Return codes The Thermostat cluster functions use the ZCL return codes defined in Section 7.2.

Parent topic:*Thermostat Cluster*

Enumerations

'Attribute ID' Enumerations The following structure contains the enumerations used to identify the attributes of the Thermostat cluster.

```
typedef enum
{
    E_CLD_THERMOSTAT_ATTR_ID_LOCAL_TEMPERATURE  = 0x0000,
    E_CLD_THERMOSTAT_ATTR_ID_OUTDOOR_TEMPERATURE,
    E_CLD_THERMOSTAT_ATTR_ID_OCCUPANCY,
    E_CLD_THERMOSTAT_ATTR_ID_ABS_MIN_HEAT_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_ABS_MAX_HEAT_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_ABS_MIN_COOL_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_ABS_MAX_COOL_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_PI_COOLING_DEMAND,
    E_CLD_THERMOSTAT_ATTR_ID_PI_HEATING_DEMAND,
    E_CLD_THERMOSTAT_ATTR_ID_LOCAL_TEMPERATURE_CALIBRATION  = 0x0010,
    E_CLD_THERMOSTAT_ATTR_ID_OCCUPIED_COOLING_SETPOINT,
    E_CLD_THERMOSTAT_ATTR_ID_OCCUPIED_HEATING_SETPOINT,
    E_CLD_THERMOSTAT_ATTR_ID_UNOCCUPIED_COOLING_SETPOINT,
    E_CLD_THERMOSTAT_ATTR_ID_UNOCCUPIED_HEATING_SETPOINT,
    E_CLD_THERMOSTAT_ATTR_ID_MIN_HEAT_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_MAX_HEAT_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_MIN_COOL_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_MAX_COOL_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_MIN_SETPOINT_DEAD_BAND,
    E_CLD_THERMOSTAT_ATTR_ID_REMOTE_SENSING,
    E_CLD_THERMOSTAT_ATTR_ID_CONTROL_SEQUENCE_OF_OPERATION,
    E_CLD_THERMOSTAT_ATTR_ID_SYSTEM_MODE,
    E_CLD_THERMOSTAT_ATTR_ID_ALARM_MASK
} teCLD_Thermostat_AttributeID;
```

Parent topic: Enumerations

'Operating Capabilities' Enumerations The following enumerations are used to set the optional attribute eControlSequenceOfOperation in the Thermostat cluster structure tsCLD_Thermostat.

```
typedef enum
{
    E_CLD_THERMOSTAT_CSOO_COOLING_ONLY = 0x00,
    E_CLD_THERMOSTAT_CSOO_COOLING_WITH_REHEAT,
    E_CLD_THERMOSTAT_CSOO_HEATING_ONLY,
    E_CLD_THERMOSTAT_CSOO_HEATING_WITH_REHEAT,
    E_CLD_THERMOSTAT_CSOO_COOLING_AND_HEATING_4_PIPES,
    E_CLD_THERMOSTAT_CSOO_COOLING_AND_HEATING_4_PIPES_WITH_REHEAT,
}teCLD_Thermostat_ControlSequenceOfOperation;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_THERMOSTAT_CSOO_COOLING_ONLY	Heat and Emergency Heating are not

possible

| |E_CLD_THERMOSTAT_CSOO_COOLING_WITH_REHEAT|Heat and Emergency Heating are not

possible

| |E_CLD_THERMOSTAT_CSOO_HEATING_ONLY|Cool and Pre-cooling are not possible| |E_CLD_THERMOSTAT_CSOO_HEATING_WITH_REHEAT|Cool and Pre-cooling are not possible| |E_CLD_THERMOSTAT_CSOO_COOLING_AND_HEATING_

4_PIPES

|All modes are possible| |E_CLD_THERMOSTAT_CSOO_COOLING_AND_HEATING_

4_PIPES_WITH_REHEAT

|All modes are possible|

Parent topic: Enumerations

'Command ID' Enumerations The following enumeration is used to specify the type of command sent to a Thermostat cluster server.

```
typedef enum
{
  E_CLD_THERMOSTAT_CMD_SETPOINT_RAISE_LOWER = 0x00,
} teCLD_Thermostat_Command;
```

The above enumerations are described in the table below.

Enumeration	Command
E_CLD_THERMOSTAT_CMD_SETPOINT_RAISE_LOWER	Setpoint Raise Or Lower

Parent topic: Enumerations

'Setpoint Raise Or Lower' Enumerations The following enumerations are used to specify an operating mode (heating, cooling or both) or the Thermostat.

```
{
   E_CLD_THERMOSTAT_SRLM_HEAT = 0x00,
   E_CLD_THERMOSTAT_SRLM_COOL,
   E_CLD_THERMOSTAT_SRLM_BOTH
}teCLD_Thermostat_SetpointRaiseOrLowerMode;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_THERMOSTAT_SRLM_HEAT	Heating mode
E_CLD_THERMOSTAT_SRLM_COOL	Cooling mode
E_CLD_THERMOSTAT_SRLM_BOTH	Heating and Cooling modes

Parent topic: Enumerations

Parent topic:*Thermostat Cluster*

Structures

Custom Data Structure The Thermostat cluster requires extra storage space to be allocated for use by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
   tsZCL_ReceiveEventAddress          sReceiveEventAddress;
   tsZCL_CallBackEvent                sCustomCallBackEvent;
   tsCLD_ThermostatCallBackMessage    sCallBackMessage;
```

(continues on next page)

(continued from previous page)

```
} tsCLD_ThermostatCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

tsCLD_ThermostatCallBackMessage For a Thermostat cluster event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_ThermostatCallBackMessage structure:

```
typedef struct
{
    uint8                                   u8CommandId;
    union
    {
      tsCLD_Thermostat_SetpointRaiseOrLowerPayload *psSetpointRaiseOrLowerPayload;
    } uMessage;
} tsCLD_ThermostatCallBackMessage;
```

where:

- u``8CommandId indicates the type of Thermostat cluster command that has been received - there is only one possibility: E_CLD_THERMOSTAT_CMD_SETPOINT_RAISE_LOWER
- u``Message is a union containing the command payload in the following form: psSetpointRaiseOrLowerPayload is a pointer to a structure containing the payload of a 'Setpoint Raise Or Lower' command - see Section 33.9.3.

Parent topic: Structures

tsCLD_Thermostat_SetpointRaiseOrLowerPayload This stucture contains the payload of a 'Setpoint Raise Or Lower' command (from the cluster client) which requests a change the value of the attribute i16OccupiedHeatingSetpoint and/or the attribute i16OccupiedCoolingSetpoint.

```
typedef struct
{
    zenum8                  eMode;
    zint8                 i8Amount;
}tsCLD_Thermostat_SetpointRaiseOrLowerPayload;
```

where:

- eMode indicates the Thermostat operating mode to which the command relates, one of:
  - E_CLD_THERMOSTAT_SRLM_HEAT (Heating)
  - E_CLD_THERMOSTAT_SRLM_COOL (Cooling)
  - E_CLD_THERMOSTAT_SRLM_BOTH (Heating and Cooling)
- i8Amount represents the value (in two's complement form) by which the setpoint corresponding to the specified operating mode is to be changed

Parent topic: Structures

Parent topic:*Thermostat Cluster*

Compile-time options To enable the Thermostat cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_THERMOSTAT
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define THERMOSTAT_SERVER
#define THERMOSTAT_CLIENT
```

Optional Attributes The optional attributes for the Thermostat cluster (see Section 33.2) are enabled by defining:

- CLD_THERMOSTAT_ATTR_ID_LOCAL_TEMPERATURE
- CLD_THERMOSTAT_ATTR_ID_OUTDOOR_TEMPERATURE
- CLD_THERMOSTAT_ATTR_ID_OCCUPANCY
- CLD_THERMOSTAT_ATTR_ID_ABS_MIN_HEAT_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_ABS_MAX_HEAT_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_ABS_MIN_COOL_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_ABS_MAX_COOL_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_PI_COOLING_DEMAND
- CLD_THERMOSTAT_ATTR_ID_PI_HEATING_DEMAND
- CLD_THERMOSTAT_ATTR_ID_LOCAL_TEMPERATURE_CALIBRATION
- CLD_THERMOSTAT_ATTR_ID_OCCUPIED_COOLING_SETPOINT
- CLD_THERMOSTAT_ATTR_ID_OCCUPIED_HEATING_SETPOINT
- CLD_THERMOSTAT_ATTR_ID_UNOCCUPIED_COOLING_SETPOINT
- CLD_THERMOSTAT_ATTR_ID_UNOCCUPIED_HEATING_SETPOINT
- CLD_THERMOSTAT_ATTR_ID_MIN_HEAT_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_MAX_HEAT_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_MIN_COOL_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_MAX_COOL_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_MIN_SETPOINT_DEAD_BAND
- CLD_THERMOSTAT_ATTR_ID_REMOTE_SENSING
- CLD_THERMOSTAT_ATTR_ID_CONTROL_SEQUENCE_OF_OPERATION
- CLD_THERMOSTAT_ATTR_ID_SYSTEM_MODE
- CLD_THERMOSTAT_ATTR_ID_ALARM_MASK

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_THERMOSTAT_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_THERMOSTAT_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Minimum Cooling Setpoint The value of the attribute i16MinCoolSetpointLimit can be set as follows:

```
#define CLD_THERMOSTAT_MIN_COOLING_SETPOINT        n
```

where n is the value to be set (in two's complement form). The default value is 0x954D.

Maximum Cooling Setpoint The value of the attribute i16MaxCoolSetpointLimit can be set as follows:

```
#define CLD_THERMOSTAT_MAX_COOLING_SETPOINT n
```

where n is the value to be set (in two's complement form). The default value is 0x7FFF.

Minimum Heating Setpoint The value of the attribute i16MinHeatSetpointLimit can be set as follows:

```
#define CLD_THERMOSTAT_MIN_HEATING_SETPOINT n
```

where n is the value to be set (in two's complement form). The default value is 0x954D.

Maximum Heating Setpoint The value of the attribute i16MaxHeatSetpointLimit can be set as follows:

```
#define CLD_THERMOSTAT_MAX_HEATING_SETPOINT n
```

where n is the value to be set (in two's complement form). The default value is 0x7FFF.

Parent topic:*Thermostat Cluster*

Fan Control Cluster This chapter describes the Fan Control cluster which is defined in the ZCL.

The Fan Control cluster has a Cluster ID of 0x0202.

Overview The Fan Control cluster is used to control the speed of a fan which may be part of a heating or cooling system. It allows the speed or state of the fan to be set, as well as the possible speeds/states that a thermostat can set.

To use the functionality of this cluster, you must include the file **FanControl.h** in your application and enable the cluster by defining CLD_FAN_CONTROL in the **zcl_options.h** file.

A Fan Control cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Fan Control cluster are fully detailed in Section 34.6.

Parent topic:*Fan Control Cluster*

Fan Control Structure and Attributes The structure definition for the Fan Control cluster is shown below.

```
typedef struct
{
    #ifdef FAN_CONTROL_SERVER
        zenum8  e8FanMode;
        zenum8  e8FanModeSequence;
    #endif
    zuint16 u16ClusterRevision;
} tsCLD_FanControl;
```

where:

- e8FanMode is a server attribute that represents the current speed/state of the fan. The attribute can be set to one of the enumerated values listed in Section34.5.2, representing off, low, medium, high, on, auto or smart.
- e8FanModeSequence is a server attribute that specifies the possible fan speeds/states that a thermostat can set. The attribute can be set to one of the enumerated values listed in Section 34.5.3, each representing a set of possible fan speeds/states.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. For cluster specifications that pre-date the ZCL r6, this attribute is set to 0.

Parent topic:*Fan Control Cluster*

Initialisation The function **eCLD_CreateFanControl()** is used to create an instance of the Fan Control cluster. The function is generally called by the initialisation function for the host device.

Parent topic:*Fan Control Cluster*

Functions The following Fan Control cluster function is provided in the NXP implementation of the ZCL:

- **Function** **Page**
- eCLD_CreateFanControl 747

eCLD_CreateFanControl

```
teZCL_Status eCLD_CreateFanControl(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Fan Control cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Fan Control cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the application profile has been initialized.

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Fan Control cluster. This parameter can refer to a pre-filled structure called tsCLD_FanControl which is provided in the **FanControl.h** file.
- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_FanControl which defines the attributes of the Fan Control cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: Functions

Parent topic:*Fan Control Cluster*

Enumerations

teCLD_FanControl_AttributeID The following structure contains the enumerations used to identify the server attributes of the Fan Control cluster.

```
typedef enum
{
    E_CLD_FAN_CONTROL_ATTR_ID_FAN_MODE              = 0x0000,
    E_CLD_FAN_CONTROL_ATTR_ID_FAN_MODE_SEQUENCE,
```

(continues on next page)

(continued from previous page)

```
} teCLD_FanControl_AttributeID;
```

Parent topic: Enumerations

teCLD_FC_FanMode The following structure contains the enumerations used to set the value of the e8FanMode attribute in the tsCLD_FanControl structure (see Section 34.2).

```
typedef enum
{
    E_CLD_FC_FAN_MODE_OFF        = 0x00,
    E_CLD_FC_FAN_MODE_LOW,       //0x01
    E_CLD_FC_FAN_MODE_MEDIUM,    //0x02
    E_CLD_FC_FAN_MODE_HIGH,      //0x03
    E_CLD_FC_FAN_MODE_ON,        //0x04
    E_CLD_FC_FAN_MODE_AUTO,      //0x05
    E_CLD_FC_FAN_MODE_SMART,     //0x06
} teCLD_FC_FanMode;
```

The above enumerations are described in the table below.

Enumeration	Description (Fan State/Speed)
E_CLD_FC_FAN_MODE_OFF	Off
E_CLD_FC_FAN_MODE_LOW	Low
E_CLD_FC_FAN_MODE_MEDIUM	Medium
E_CLD_FC_FAN_MODE_HIGH	High
E_CLD_FC_FAN_MODE_ON	On
E_CLD_FC_FAN_MODE_AUTO	Auto (fan speed is self-regulated)
E_CLD_FC_FAN_MODE_SMART	Smart (when the space is occupied, the fan is always on)

Parent topic: Enumerations

teCLD_FC_FanModeSequence The following structure contains the enumerations used to set the value of the e8FanModeSequence attribute in the tsCLD_FanControl structure (see Section34.2).

```
typedef enum
{
    E_CLD_FC_FAN_MODE_SEQUENCE_LOW_MED_HIGH        = 0x00,
    E_CLD_FC_FAN_MODE_SEQUENCE_LOW_HIGH,           //0x01
    E_CLD_FC_FAN_MODE_SEQUENCE_LOW_MED_HIGH_AUTO,  //0x02
    E_CLD_FC_FAN_MODE_SEQUENCE_LOW_HIGH_AUTO,      //0x03
    E_CLD_FC_FAN_MODE_SEQUENCE_ON_AUTO,            //0x04
} teCLD_FC_FanModeSequence;
```

The above enumerations are described in the table below (the fan speeds/states refer to those listed in Section 34.5.2).

Enumeration	Description (Set of Fan Speeds/States)
E_CLD_FC_FAN_MODE_SEQUENCE_LOW_MED_HIGH	Low/Med/High
E_CLD_FC_FAN_MODE_SEQUENCE_LOW_HIGH	Low/High
E_CLD_FC_FAN_MODE_SEQUENCE_LOW_MED_HIGH_A	Low/Med/High/Auto
E_CLD_FC_FAN_MODE_SEQUENCE_LOW_HIGH_AUTO	Low/High/Auto
E_CLD_FC_FAN_MODE_SEQUENCE_ON_AUTO	On/Auto

Parent topic: Enumerations

Parent topic:*Fan Control Cluster*

Compile-time options To enable the Fan Control cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_FAN_CONTROL
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define FAN_CONTROL_CLIENT
#define FAN_CONTROL_SERVER
```

Parent topic:*Fan Control Cluster*

Thermostat UI Configuration Cluster This chapter outlines the Thermostat User Interface (UI) Configuration cluster which is defined in the ZCL and provides an interface for configuring the user interface (keypad and/or LCD screen) of a thermostat - this interface may be located on a controlling device which is remote from the thermostat.

The Thermostat UI Configuration cluster has a Cluster ID of 0x0204.

Overview The Thermostat UI Configuration cluster is required in ZigBee devices as indicated in the table below.

	Server-side	Client-side
Mandatory in...		
Optional in...	Thermostat	Configuration Tool

Combined Interface

|

The Thermostat UI Configuration cluster is enabled by defining CLD_THERMOSTAT_UI_CONFIG in the **zcl_options.h** file.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Thermostat UI Configuration cluster are fully detailed in Section 35.7.

Parent topic:*Thermostat UI Configuration Cluster*

Cluster structure and attributes The Thermostat UI Configuration cluster is contained in the following tsCLD_ThermostatUIConfig structure:

```
typedef struct
{
#ifdef THERMOSTAT_UI_CONFIG_SERVER
    zenum8          eTemperatureDisplayMode;
    zenum8          eKeypadLockout;
#endif
    zuint16         u16ClusterRevision;
} tsCLD_ThermostatUIConfig;;
```

where:

- eTemperatureDisplayMode specifies the units (Celsius or Fahrenheit) used to display temperature on the screen of the user interface. Enumerations are provided:
  - E_CLD_THERMOSTAT_UI_CONFIG_TEMPERATURE_DISPLAY_MODE_CELSIUS
  - E_CLD_THERMOSTAT_UI_CONFIG_TEMPERATURE_DISPLAY_MODE_FAHRENHEIT
- eKeypadLockout specifies the level of functionality that is available via the keypad of the user interface. Enumerations are provided:
  - E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_NO_LOCKOUT
  - E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_1_LOCKOUT
  - E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_2_LOCKOUT
  - E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_3_LOCKOUT
  - E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_4_LOCKOUT
  - E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_5_LOCKOUT
    - The functionality of each level is manufacturer-defined but level 5 represents the minimum functionality.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Thermostat UI Configuration Cluster*

Initialization The function **eCLD_ThermostatUIConfigCreateThermostatUIConfig()** is used to create an instance of the Thermostat UI Configuration cluster. The function is generally called by the initialization function for the host device.

Parent topic:*Thermostat UI Configuration Cluster*

Functions The following Thermostat UI Configuration cluster functions are provided:

- **Function** **Page**
- eCLD_ThermostatUIConfigCreateThermostatUIConfig 754
- eCLD_ThermostatUIConfigConvertTemp 756

eCLD_ThermostatUIConfigCreateThermostatUIConfig

```
teZCL_Status eCLD_ThermostatUIConfigCreateThermostatUIConfig(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    sZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Thermostat UI Configuration cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Thermostat UI Configuration cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device (for example, the Thermostat device) is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be the first Thermostat UI Configuration cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type uint8) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Thermostat UI Configuration cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* : Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Thermostat UI Configuration cluster. This parameter can refer to a pre-filled structure called sCLD_``Thermostat``UIConfig which is provided in the **ThermostatUIConfig.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_``Thermostat``UIConfig which defines the attributes of Thermostat UI Configuration cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS

- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_ThermostatUIConfigConvertTemp

```
teZCL_Status eCLD_ThermostatUIConfigConvertTemp(
    uint8 u8SourceEndPointId,
    bool bConvertCToF,
    int16 *pi16Temperature);
```

Description This function can be used on a Thermostat UI Configuration cluster server to convert a temperature from units of Celsius to Fahrenheit or vice-versa (the direction must be specified). The temperature value to be converted is provided to the function as a pointer to a memory location where the input value is stored. This stored value is replaced with the converted temperature value by the function (over-writing the input value).

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the Thermostat UI Configuration cluster resides
- *bConvertCToF*: Direction of temperature conversion:
- TRUE - Celsius to Fahrenheit
- FALSE - Fahrenheit to Celsius
- *pi16Temperature*: Pointer to location containing the temperature value to be converted. The converted temperature value is also output to this location by the function

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_INVALID_VALUE
- E_ZCL_DENY_ATTRIBUTE_ACCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND

Parent topic: Functions

Parent topic:*Thermostat UI Configuration Cluster*

Return codes The Thermostat UI Configuration cluster functions use the ZCL return codes defined in Section 7.2.

Parent topic:*Thermostat UI Configuration Cluster*

Enumerations

'Attribute ID' Enumerations The following structure contains the enumerations used to identify the attributes of the Thermostat UI Configuration cluster.

```
typedef enum
{
  E_CLD_THERMOSTAT_UI_CONFIG_ATTR_ID_TEMPERATURE_DISPLAY_MODE = 0x0000
  E_CLD_THERMOSTAT_UI_CONFIG_ATTR_ID_KEYPAD_LOCKOUT
} teCLD_ThermostatUIConfig_AttributeID;
```

Parent topic: Enumerations

'Temperature Display Mode' Enumerations The following enumerations are used to set the optional attribute eTemperatureDisplayMode in the Thermostat UI Configuration cluster structure tsCLD_ThermostatUIConfig.

```
typedef enum
{
  E_CLD_THERMOSTAT_UI_CONFIG_TEMPERATURE_DISPLAY_MODE_CELSIUS = 0x00,
  E_CLD_THERMOSTAT_UI_CONFIG_TEMPERATURE_DISPLAY_MODE_FAHRENHEIT
} teCLD_ThermostatUIConfig_TemperatureDisplay;
```

The above enumerations represent the units of temperature available to display temperature on the screen of the user interface and are described in the table below.

Enumeration	Description
E_CLD_THERMOSTAT_UI_CONFIG_TEMPERATURE_DISPLAY_MO	Display temperature in Celsius
E_CLD_THERMOSTAT_UI_CONFIG_TEMPERATURE_DISPLAY_MO	Display temperature in Fahrenheit

Parent topic: Enumerations

'Keypad Functionality' Enumerations The following enumeration is used to set the optional attribute eKeypadLockout in the Thermostat UI Configuration cluster structure tsCLD_ThermostatUIConfig.

```
typedef enum
{
   E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_NO_LOCKOUT = 0x00,
   E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_1_LOCKOUT,
   E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_2_LOCKOUT,
   E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_3_LOCKOUT,
   E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_4_LOCKOUT,
   E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_5_LOCKOUT
} teCLD_ThermostatUIConfig_KeyPadLockout;
```

The above enumerations represent levels of functionality available via the keypad of the user interface. The functionality of each level is manufacturer-defined but level 5 represents the minimum functionality.

Parent topic: Enumerations

Parent topic:*Thermostat UI Configuration Cluster*

Compile-time Options To enable the Thermostat UI Configuration cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_THERMOSTAT_UI_CONFIG
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define THERMOSTAT_UI_CONFIG_SERVER
#define THERMOSTAT_UI_CONFIG_CLIENT
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_THERMOSTAT_UI_CONFIG_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Thermostat UI Configuration Cluster*

Part VII: Closure Clusters This part comprises one chapter:

- *Chapter 36* details the **Door Lock** cluster

Door Lock Cluster This chapter outlines the Door Lock cluster, which provides an interface to a set values representing the state of a door lock and (optionally) the door.

The Door Lock cluster has a Cluster ID of 0x0101.

Overview The Door Lock cluster is required in ZigBee devices as indicated in the table below.

	Server-side	Client-side
Mandatory in...	Door Lock	Door Lock Controller
Optional in...		Remote Control

The Door Lock cluster is enabled by defining CLD_DOOR_LOCK in the **zcl_options.h** file.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Door Lock cluster are fully detailed in Section 36.9.

Parent topic:*Door Lock Cluster*

Door Lock Cluster structure and attributes The Door Lock cluster is contained in the following ts``CLD_DoorLock structure:

```
typedef struct
{
#ifdef DOOR_LOCK_SERVER
    zenum8                  eLockState;
```

(continues on next page)

(continued from previous page)

```
    zenum8                  eLockType;
    zbool                   bActuatorEnabled;
#ifdef CLD_DOOR_LOCK_ATTR_DOOR_STATE
    zenum8                  eDoorState;
#endif
#ifdef CLD_DOOR_LOCK_ATTR_NUMBER_OF_DOOR_OPEN_EVENTS
    zuint32                 u32NumberOfDoorOpenEvent;
#endif
#ifdef CLD_DOOR_LOCK_ATTR_NUMBER_OF_DOOR_CLOSED_EVENTS
    zuint32                 u32NumberOfDoorClosedEvent;
#endif
#ifdef CLD_DOOR_LOCK_ATTR_NUMBER_OF_MINUTES_DOOR_OPENED
    zuint16                 u16NumberOfMinutesDoorOpened;
#endif
#ifdef CLD_DOOR_LOCK_ZIGBEE_SECUTRITY_LEVEL
    zuint8                  u8ZigbeeSecurityLevel;
#endif
#ifdef CLD_DOOR_LOCK_ATTRIBUTE_REPORTING_STATUS
    zuint8                  u8AttributeReportingStatus;
#endif
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_DoorLock;
```

where:

- eLockState is a mandatory attribute indicating the state of the lock, one of:
  - E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED
  - E_CLD_DOORLOCK_LOCK_STATE_LOCK
  - E_CLD_DOORLOCK_LOCK_STATE_UNLOCK
- eLockType is a mandatory attribute representing the type of door lock, one of:
  - E_CLD_DOORLOCK_LOCK_TYPE_DEAD_BOLT
  - E_CLD_DOORLOCK_LOCK_TYPE_MAGNETIC
  - E_CLD_DOORLOCK_LOCK_TYPE_OTHER
  - E_CLD_DOORLOCK_LOCK_TYPE_MORTISE
  - E_CLD_DOORLOCK_LOCK_TYPE_RIM
  - E_CLD_DOORLOCK_LOCK_TYPE_LATCH_BOLT
  - E_CLD_DOORLOCK_LOCK_TYPE_CYLINDRICAL_LOCK
  - E_CLD_DOORLOCK_LOCK_TYPE_TUBULAR_LOCK
  - E_CLD_DOORLOCK_LOCK_TYPE_INTERCONNECTED_LOCK
  - E_CLD_DOORLOCK_LOCK_TYPE_DEAD_LATCH
  - E_CLD_DOORLOCK_LOCK_TYPE_DOOR_FURNITURE
- bActuatorEnabled is a mandatory attribute indicating whether the actuator for the door lock is enabled:
  - TRUE - enabled
  - FALSE - disabled
- eDoorState is an optional attribute indicating the current state of the door, one of:
  - E_CLD_DOORLOCK_DOOR_STATE_OPEN

  - E_CLD_DOORLOCK_DOOR_STATE_CLOSED
  - E_CLD_DOORLOCK_DOOR_STATE_ERROR_JAMMED
  - E_CLD_DOORLOCK_DOOR_STATE_ERROR_FORCED_OPEN
  - E_CLD_DOORLOCK_DOOR_STATE_ERROR_UNSPECIFIED
- u32NumberOfDoorOpenEvent is an optional attribute representing the number of 'door open' events that have occurred
- u32NumberOfDoorClosedEvent is an optional attribute representing the number of 'door close' events that have occurred
- u16NumberOfMinutesDoorOpened is an optional attribute representing the length of time, in minutes, that the door has been open since the last 'door open' event
- u8ZigbeeSecurityLevel is an optional attribute representing the ZigBee PRO security level that should be applied to communications between a cluster server and client:
  - 0: Network-level security only
  - 1 or higher: Application-level security (in addition to Network-level security)
    * Application-level security is an enhancement to the Door Lock cluster and is currently not certifiable.

Note: The application must not write directly to the u8ZigbeeSecurityLevel attribute. If required, Application-level security should be enabled only using the function **eCLD_DoorLockSetSecurityLevel()**. For more information, refer to the description of this function on page 771.

```
u8AttributeReportingStatus is an optional attribute that should be enabled when attribute report-
ing is used for the cluster (see [Section 2.3.5](../../ZCL_fundamentals/topics/attribute_reporting.md#id_
↪d3e2d682-d5e6-4aa5-870f-4d2e0320637e)). The value of this attribute indicates whether there are at-
tribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved.
↪ This attribute is also described in [Section 2.4](../../ZCL_fundamentals/topics/global_attributes.md#id_
↪8f80e769-8ccd-4b4f-8609-579f4457d527).
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specifica-
tion on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster re-
vision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This at-
tribute is also described in [Section 2.4](../../ZCL_fundamentals/topics/global_attributes.md#id_8f80e769-
↪8ccd-4b4f-8609-579f4457d527).
```

Parent topic:*Door Lock Cluster*

Attributes for Default Reporting The following attributes of the Door Lock cluster can be selected for default reporting:

```
eLockState
eDoorState
```

Attribute reporting (including default reporting) is described in *Appendix B.* Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic:*Door Lock Cluster*

Door Lock Events The Door Lock cluster has its own events that are handled through the callback mechanism outlined in *Chapter 3*. If a device uses the Door Lock cluster then Door Lock event handling must be included in the callback function for the associated endpoint, where this

callback function is registered through the relevant endpoint registration function (for example, through **eHA_RegisterDoorLockEndPoint()** for a Door Lock device). The relevant callback function will then be invoked when a Door Lock event occurs.

For a Door Lock event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_DoorLockCallBackMessage structure:

```
typedef struct
{
    uint8    u8CommandId;
    union
    {
      tsCLD_DoorLock_LockUnlockResponsePayload *psLockUnlockResponsePayload;
    }uMessage;
 }tsCLD_DoorLockCallBackMessage;
```

When a Door Lock event occurs, one of two command types could have been received. The relevant command type is specified through the u8CommandId field of the tsCLD_DoorLockCallBackMessage structure. The possible command types are detailed below.

u8CommandId Enumeration	Description
E_CLD_DOOR_LOCK_CMD_LOCK	A lock request command has been received by the clus-ter server
E_CLD_DOOR_LOCK_CMD_UNLC	An unlock request command has been received by the cluster server

Parent topic:*Door Lock Cluster*

Functions The following Door Lock cluster functions are provided:

- **Function Page**
- eCLD_DoorLockCreateDoorLock 766
- eCLD_DoorLockSetLockState 768
- eCLD_DoorLockGetLockState 769
- eCLD_DoorLockCommandLockUnlockRequestSend 770
- eCLD_DoorLockSetSecurityLevel 771

eCLD_DoorLockCreateDoorLock

```
teZCL_Status eCLD_DoorLockCreateDoorLock(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description This function creates an instance of the Door Lock cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Door Lock cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device (e.g. the Door Lock device) will be used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be the first Door Lock cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type uint8) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Door Lock cluster.

The function initializes the array elements to zero.

Parameters

- **psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- **psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Door Lock cluster. This parameter can refer to a pre-filled structure called sCLD_DoorLock which is provided in the **DoorLock.h** file.
- **pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_DoorLock which defines the attributes of Door Lock cluster. The function initializes the attributes with default values.
- **pu8AttributeControlBits* : Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_DoorLockSetLockState

```
teZCL_Status eCLD_DoorLockSetLockState(
    uint8 u8SourceEndPointId,
    teCLD_DoorLock_LockState eLock);
```

Description This function can be used on a Door Lock cluster server to set the value of the eLockState attribute which represents the current state of the door lock (locked, unlocked or not fully locked).

Depending on the specified value of *eLock*, the attribute will be set to one of the following:

- E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED
- E_CLD_DOORLOCK_LOCK_STATE_LOCK
- E_CLD_DOORLOCK_LOCK_STATE_UNLOCK

This function generates an update event to inform the application when the change has been made.

Parameters

- *u8SourceEndPointId* Number of the endpoint on which the Door Lock cluster resides
- *eLock* State in which to put the door lock, one of:
- E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED
- E_CLD_DOORLOCK_LOCK_STATE_LOCK
- E_CLD_DOORLOCK_LOCK_STATE_UNLOCK

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Functions

eCLD_DoorLockGetLockState

```
teZCL_Status eCLD_DoorLockGetLockState(
    uint8 u8SourceEndPointId,
    teCLD_DoorLock_LockState *peLock);
```

Description This function can be used on a Door Lock cluster server to obtain the value of the eLockState attribute which represents the current state of the door lock (locked, unlocked or not fully locked).

The value of the attribute is returned through the location pointed to by peLock and can be any one of the following:

- E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED
- E_CLD_DOORLOCK_LOCK_STATE_LOCK
- E_CLD_DOORLOCK_LOCK_STATE_UNLOCK

Parameters

- *u8SourceEndPointId* Number of the endpoint on which the Door Lock cluster resides
- *peLock* Pointer to location to receive the obtained state of the door lock, which will be one of:
- E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED

- E_CLD_DOORLOCK_LOCK_STATE_LOCK
- E_CLD_DOORLOCK_LOCK_STATE_UNLOCK

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Functions

eCLD_DoorLockCommandLockUnlockRequestSend

```
teZCL_Status eCLD_DoorLockCommandLockUnlockRequestSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    teCLD_DoorLock_CommandID eCommand);
```

Description This function can be used on a Door Lock cluster client to send a lock or unlock command to the Door Lock cluster server.

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId* Number of the remote endpoint to which the request is sent
- *psDestinationAddress* Pointer to a structure containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *eCommand* The command to be sent, one of:
- E_CLD_DOOR_LOCK_CMD_LOCK
- E_CLD_DOOR_LOCK_CMD_UNLOCK

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Functions

eCLD_DoorLockSetSecurityLevel

```
teZCL_Status eCLD_DoorLockSetSecurityLevel(
    uint8 u8SourceEndPointId,
    bool bServer,
    uint8 u8SecurityLevel);
```

Description This function can be used to set the level of security to be used by the Door Lock cluster: Network-level security or Application-level security. By default, only Network-level security is implemented, but this function can be used to enable Application-level security (in addition to Network-level security). For more information on ZigBee security, refer to the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

Application-level security is an enhancement to the Door Lock cluster and is currently not certifiable. It is enabled through an optional attribute of the cluster, but the application must not write directly to this attribute - if required, Application-level security should be enabled only using this function.

To use Application-level security, it is necessary to call this function on the Door Lock cluster server and client nodes. If an application link key is to be used which is not the default one, the new link key must be subsequently specified on both nodes using the ZigBee PRO function **ZPS_eAplZdoAddReplaceLinkKey()**.

Parameters

- *u8SourceEndPointId* Number of the local endpoint on which the Door Lock cluster resides
- *bIsServer* Type of local cluster instance (server or client):
- TRUE - server
- FALSE - client
- *u8SecurityLevel* The security level to be set:
- 0: Network-level security only
- 1 or higher: Application-level security

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Functions

Parent topic:*Door Lock Cluster*

Return codes The Door Lock cluster functions use the ZCL return codes defined in Section 7.2.

Parent topic:*Door Lock Cluster*

Enumerations

'Attribute ID' Enumerations The following structure contains the enumerations used to identify the attributes of the Door Lock cluster.

```
typedef enum
{
    E_CLD_DOOR_LOCK_ATTR_ID_LOCK_STATE = 0x0000,
    E_CLD_DOOR_LOCK_ATTR_ID_LOCK_TYPE,
    E_CLD_DOOR_LOCK_ATTR_ID_ACTUATOR_ENABLED,
    E_CLD_DOOR_LOCK_ATTR_ID_DOOR_STATE,
    E_CLD_DOOR_LOCK_ATTR_ID_NUMBER_OF_DOOR_OPEN_EVENTS,
    E_CLD_DOOR_LOCK_ATTR_ID_NUMBER_OF_DOOR_CLOSED_EVENTS,
    E_CLD_DOOR_LOCK_ATTR_ID_NUMBER_OF_MINUTES_DOOR_OPENED,
    E_CLD_DOOR_LOCK_ATTR_ID_ZIGBEE_SECURITY_LEVEL = 0x0034
} teCLD_DoorLock_Cluster_AttrID;
```

Parent topic: Enumerations

'Lock State' Enumerations The following enumerations are used to set the eLockState element in the Door Lock cluster structure tsCLD_DoorLock.

```
typedef enum
{
    E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED  = 0x00,
    E_CLD_DOORLOCK_LOCK_STATE_LOCKED,
    E_CLD_DOORLOCK_LOCK_STATE_UNLOCKED,
    E_CLD_DOORLOCK_LOCK_STATE_UNDEFINED         = 0xFF
} teCLD_DoorLock_LockState;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED	Not fully locked
E_CLD_DOORLOCK_LOCK_STATE_LOCK	Locked
E_CLD_DOORLOCK_LOCK_STATE_UNLOCK	Unlocked

Parent topic: Enumerations

'Lock Type' Enumerations The following enumerations are used to set the eLockType element in the Door Lock cluster structure tsCLD_DoorLock.

```
typedef enum
{
    E_CLD_DOORLOCK_LOCK_TYPE_DEAD_BOLT  = 0x00,
    E_CLD_DOORLOCK_LOCK_TYPE_MAGNETIC,
    E_CLD_DOORLOCK_LOCK_TYPE_OTHER,
    E_CLD_DOORLOCK_LOCK_TYPE_MORTISE,
    E_CLD_DOORLOCK_LOCK_TYPE_RIM,
    E_CLD_DOORLOCK_LOCK_TYPE_LATCH_BOLT,
    E_CLD_DOORLOCK_LOCK_TYPE_CYLINDRICAL_LOCK,
    E_CLD_DOORLOCK_LOCK_TYPE_TUBULAR_LOCK,
    E_CLD_DOORLOCK_LOCK_TYPE_INTERCONNECTED_LOCK,
    E_CLD_DOORLOCK_LOCK_TYPE_DEAD_LATCH,
    E_CLD_DOORLOCK_LOCK_TYPE_DOOR_FURNITURE
} teCLD_DoorLock_LockType;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_DOORLOCK_LOCK_TYPE_DEAD_BOLT	Dead bold lock
E_CLD_DOORLOCK_LOCK_TYPE_MAGNETIC	Magnetic lock
E_CLD_DOORLOCK_LOCK_TYPE_OTHER	Other type of lock
E_CLD_DOORLOCK_LOCK_TYPE_MORTISE	Mortise lock
E_CLD_DOORLOCK_LOCK_TYPE_RIM	Rim lock
E_CLD_DOORLOCK_LOCK_TYPE_LATCH_BOLT	Latch bolt
E_CLD_DOORLOCK_LOCK_TYPE_CYLINDRICAL_LOCK	Cylindrical lock
E_CLD_DOORLOCK_LOCK_TYPE_TUBULAR_LOCK	Tubular lock
E_CLD_DOORLOCK_LOCK_TYPE_INTERCONNECTED_LOCK	Interconnected lock
E_CLD_DOORLOCK_LOCK_TYPE_DEAD_LATCH	Dead latch
E_CLD_DOORLOCK_LOCK_TYPE_DOOR_FURNITURE	Door furniture lock

Parent topic: Enumerations

'Door State' Enumerations The following enumerations are used to set the optional eDoorState element in the Door Lock cluster structure tsCLD_DoorLock.

```
typedef enum
{
    E_CLD_DOORLOCK_DOOR_STATE_OPEN  = 0x00,
    E_CLD_DOORLOCK_DOOR_STATE_CLOSED,
    E_CLD_DOORLOCK_DOOR_STATE_ERROR_JAMMED,
    E_CLD_DOORLOCK_DOOR_STATE_ERROR_FORCED_OPEN,
    E_CLD_DOORLOCK_DOOR_STATE_ERROR_UNSPECIFIED,
    E_CLD_DOORLOCK_DOOR_STATE_UNDEFINED = 0xFF
} teCLD_DoorLock_DoorState;;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_DOORLOCK_DOOR_STATE_OPEN	Door is open
E_CLD_DOORLOCK_DOOR_STATE_CLOSED	Door is closed
E_CLD_DOORLOCK_DOOR_STATE_ERROR_JAMMED	Door is jammed
E_CLD_DOORLOCK_DOOR_STATE_ERROR_FORCED_OPEN	Door has been forced open
E_CLD_DOORLOCK_DOOR_STATE_ERROR_UNSPECIFIED	Door is in an unknown state

Parent topic: Enumerations

'Command ID' Enumerations The following enumerations are used to set specify the type of command (lock or unlock) sent to a Door Lock cluster server.

```
typedef enum
{
    E_CLD_DOOR_LOCK_CMD_LOCK
    E_CLD_DOOR_LOCK_CMD_UNLOCK
} teCLD_DoorLock_CommandID;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_DOOR_LOCK_CMD_LOCK	A lock command
E_CLD_DOOR_LOCK_CMD_UNLOCK	An unlock command

Parent topic: Enumerations

Parent topic:*Door Lock Cluster*

Structures

tsCLD_DoorLockCallBackMessage For a Door Lock event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_DoorLockCallBackMessage structure:

```
typedef struct
{
    uint8 u8CommandId;
    union
    {
      tsCLD_DoorLock_LockUnlockResponsePayload *psLockUnlockResponsePayload;
    }uMessage;
 }tsCLD_DoorLockCallBackMessage;
```

where:

- u``8CommandId indicates the type of Door Lock command (lock or unlock) that has been received, one of:
  - E_CLD_DOOR_LOCK_CMD_LOCK
  - E_CLD_DOOR_LOCK_CMD_UNLOCK
- uMessage is a union containing the command payload in the following form:
  - psLockUnlockResponsePayload is a pointer to a structure containing the response payload of the received command - see Section 36.8.2

Parent topic: Structures

tsCLD_DoorLock_LockUnlockResponsePayload This stucture contains the payload of a lock/unlock command response (from the cluster server).

```
typedef struct
{
    zenum8      eStatus;
}tsCLD_DoorLock_LockUnlockResponsePayload;
```

where e``Status indicates whether the command was received:

0x00 - SUCCESS, 0x01 - FAILURE (all other values are reserved).

Parent topic: Structures

Parent topic:*Door Lock Cluster*

Compile-time options To enable the Door Lock cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_DOOR_LOCK
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define CLD_DOOR_LOCK_SERVER
#define CLD_DOOR_LOCK_CLIENT
```

Optional Attributes Add this line to enable the optional Door State attribute:

```
#define CLD_DOOR_LOCK_ATTR_DOOR_STATE
```

Add this line to enable the optional Number Of Door Open Events attribute:

```
#define CLD_DOOR_LOCK_ATTR_NUMBER_OF_DOOR_OPEN_EVENTS
```

Add this line to enable the optional Number Of Door Closed Events attribute:

```
#define CLD_DOOR_LOCK_ATTR_NUMBER_OF_DOOR_CLOSED_EVENTS
```

Add this line to enable the optional Number Of Minutes Door Opened attribute:

```
#define CLD_DOOR_LOCK_ATTR_NUMBER_OF_MINUTES_DOOR_OPENED
```

Add this line to enable the optional ZigBee Security Level attribute:

```
#define CLD_DOOR_LOCK_ZIGBEE_SECURITY_LEVEL
```

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_DOOR_LOCK_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Global Attributes Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_DOOR_LOCK_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Door Lock Cluster*

Part VIII: Security and Safety Clusters This part comprises three chapters:

- *Chapter 37* details the **IAS Zone** cluster
- *Chapter 38* details the **IAS ACE (Ancillary Control Equipment)** cluster
- *Chapter 39* details the **IAS WD (Warning Device)** cluster

IAS Zone Cluster This chapter describes the IAS Zone cluster which provides an interface to an IAS Zone device in an IAS (Intruder Alarm System).

The IAS Zone cluster has a Cluster ID of 0x0500.

Overview The IAS Zone cluster provides an interface to an IAS Zone device, which provides security alarm triggers for a zone or region of a building (e.g. fire detection). The cluster allows an IAS Zone device to be configured/controlled from a CIE (Control and Indicating Equipment) device. The server side of the cluster is implemented on the IAS Zone device and the client side is implemented on the CIE device. The IAS Zone device is detailed in the *ZigBee Devices User Guide (JNUG3131)*.

The cluster supports the following functionality:

- Up to two alarm types per zone, Alarm1 and Alarm2
- 'Low battery' reports
- Supervision of the IAS network

To use the functionality of this cluster, you must include the file **IASZone.h** in your application and enable the cluster by defining CLD_IASZONE in the **zcl_options.h** file.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the IAS Zone cluster are fully detailed in Section 37.7.

The information that can potentially be stored in this cluster is organised into the following attribute sets:

- Zone information
- Zone settings

Parent topic:*IAS Zone Cluster*

IAS Zone Structure and Attributes The structure definition for the IAS Zone cluster is:

```
typedef struct
{
#ifdef IASZONE_SERVER
    zenum8      e8ZoneState;
    zenum16     e16ZoneType;
    zbmap16     b16ZoneStatus;
    zuint64     u64IASCIEAddress;
    zuint8      u8ZoneId;
#ifdef CLD_IASZONE_ATTR_ID_NUMBER_OF_ZONE_SENSITIVITY_LEVELS
    zuint8      u8NumberOfZoneSensitivityLevels;
#endif
#ifdef CLD_IASZONE_ATTR_ID_CURRENT_ZONE_SENSITIVITY_LEVEL
    zuint8      u8CurrentZoneSensitivityLevel;
#endif
#endif
    zuint16     u16ClusterRevision;
} tsCLD_IASZone;
```

where:

'Zone Information' Attribute Set

- e8ZoneState is a mandatory attribute which indicates the membership status of the device in an IAS system (enrolled or not enrolled) - one of:
  - E_CLD_IASZONE_STATE_NOT_ENROLLED (0x00)
  - E_CLD_IASZONE_STATE_ENROLLED (0x01)
    - 'Enrolled' means that the cluster client will react to Zone State Change Notification commands from the cluster server.
- e16ZoneType is a mandatory attribute which indicates the zone type and the types of security detectors that can trigger the alarms, Alarm1 and Alarm2:

Enumeration	Value	Type	Alarm1	Alarm2
E_CLD_IASZONE_TYPE_STANDARI	0x00(	Standard CIE	System alarm	-
E_CLD_IASZONE_TYPE_MOTION _ SENSOR	0x00(	Motion sensor	Intrusion indi- cation	Presence indica- tion
E_CLD_IASZONE_TYPE_CONTACT _ SWITCH	0x001	Contact switch	First portal open close	Second portal open-close
E_CLD_IASZONE_TYPE_FIRE_SENS	0x002	Fire sensor	Fire indication	-
E_CLD_IASZONE_TYPE_WATER_SE	0x002	Water sensor	Water overflow indication	-
E_CLD_IASZONE_TYPE_GAS_SENS	0x002	Gas sensor	Carbon monox- ide indication	Cooking indica- tion
E_CLD_IASZONE_TYPE_PERSONAI _ EMERGENCY_DEVICE	0x002	Personal emer- gency device	Fall/concussion	Emergency but- ton
E_CLD_IASZONE_TYPE_VIBRATIOI _ MOVEMENT_SENSOR	0x002	Vibration movement sen- sor	Movement indi- cation	Vibration
E_CLD_IASZONE_TYPE_REMOTE _ CONTROL	0x01(	Remote control	Panic	Emergency
E_CLD_IASZONE_TYPE_KEY_FOB	0x011	Key fob	Panic	Emergency
E_CLD_IASZONE_TYPE_KEYPAD	0x021	Keypad	Panic	Emergency
E_CLD_IASZONE_TYPE_STANDARI _ WARNING_DEVICE	0x022	Standard warn- ing device	-	-
E_CLD_IASZONE_TYPE_INVALID_Z	0xFFI	Invalid zone type	-	-

- b``16ZoneStatus is a mandatory attribute which is a 16-bit bitmap indicating the status of each of the possible notification triggers from the device:

Bit	Description
0	Alarm1:1 - Opened or alarmed0 - Closed or not alarned
1	Alarm2:1 - Opened or alarmed0 - Closed or not alarned
2	Tamper:1 - Tampered with0 - Not tampered with
3	Battery:1 - Low0 - OK
4	Supervision reports: 1 - Reports0 - No reports (See the note below)
5	Restore reports: 1 - Reports0 - No reports (See the note below)
6	Trouble:1 - Trouble/failure0 - OK
7	AC (mains): 1 - Fault0 - OK
8	Test mode:1 - Sensor in test mode0 - Sensor in operational mode
9	Battery defect: 1 - Defective battery detected0 - Battery OK
10-15	Reserved

- Bit 4 indicates whether the Zone device issues periodic Zone Status Change Notification commands that may be used by the CIE device as evidence that the Zone device is operational.

- Bit 5 indicates whether the Zone device issues a Zone Status Change Notification command to notify when an alarm is no longer present (some Zone devices do not have the ability to detect when the alarm condition has disappeared).

'Zone Settings' Attribute Set

- u64IASCIEAddress is a mandatory attribute containing the 64-bit IEEE/MAC address of the CIE device to which the cluster server must send commands/ notifications
- u8ZoneId is a mandatory attribute containing the 8-bit identifier for the zone allocated by the CIE device at the time of enrollment
- u8NumberOfZoneSensitivityLevels is an optional attribute containing the number of sensitivity levels (for the detectable quantity) for the zone - for devices that have only one sensitivity level, this attribute need not be enabled or can be set to 0x00 or 0x01

  Note: The definition of a sensitivity level is manufacturer-specific but detector 'sensitivity' should increase with higher values of this attribute.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4

Parent topic:*IAS Zone Cluster*

Enrollment An IAS Zone device hosting the IAS Zone cluster server must be paired with a CIE device hosting the cluster client. This pairing is implemented by the process of 'enrollment' which, for extra security, provides a layer of pairing in addition to ZigBee PRO binding - if required, binding is implemented as part of the enrollment process.

During enrollment, the CIE device sends its IEEE/MAC address to the Zone device as well as a Zone ID, which is a unique 8-bit identifier that the CIE device assigns to the Zone device. These values are stored in the u64IASCIEAddress and u8ZoneId attributes on the Zone device (cluster server) - see Section 37.2. In addition, once enrollment has completed, the e8ZoneState attribute is set to 'enrolled'. Subsequently, the Zone device will only communicate with the paired CIE device.

Enrollment begins just after the Zone device joins the network. This device must then periodically poll for data (from the CIE device), ideally once every 2 seconds (or faster) but no slower than once every 7 seconds. This polling must continue until the e8ZoneState attribute has been updated to 'enrolled'. However, if the IAS Zone device supports the Poll Control cluster, polling at the above rate should continue until the Poll Control cluster configuration is changed.

Three methods of enrollment are available:

- Trip-to-Pair, described in Section 37.3.1
- Auto-Enroll-Response, described in Section 37.3.2
- Auto-Enroll-Request, described in Section 37.3.3

A cluster server and client can each implement both Trip-to-Pair and Auto-Enroll-Response or just Auto-Enroll-Request.

Trip-to-Pair The Trip-to-Pair method of enrollment is described below:

1. After the IAS Zone device joins the network, the CIE device performs a service discovery.
2. If the CIE device determines that it wants to enroll the Zone device, it sends a Write Attribute command to the Zone device in order to write its IEEE/MAC address to the relevant attribute.

3. The Zone device may optionally create a binding table entry for the CIE device and store the CIE device's IEEE/MAC address there.
4. The Zone device waits for the authorization of the enrollment via a user input (for example, a button-press) and, on this input, sends a Zone Enroll Request command to the CIE device.
5. The CIE device assigns a Zone ID to the Zone device and sends a Zone Enroll Response command to it.
6. The Zone device updates its attributes to stored the assigned Zone ID and update its zone state to 'enrolled'.

Parent topic: Enrollment

Auto-Enroll-Response The Auto-Enroll-Response method of enrollment is described below:

1. After the IAS Zone device joins the network, the CIE device performs a service discovery.
2. If the CIE device determines that it wants to enroll the Zone device, it sends a Write Attribute command to the Zone device in order to write its IEEE/MAC address to the relevant attribute.
3. The Zone device may optionally create a binding table entry for the CIE device and store the CIE device's IEEE/MAC address there.
4. The CIE device assigns a Zone ID to the Zone device and sends a Zone Enroll Response command to it.
5. The Zone device updates its attributes to stored the assigned Zone ID and update its zone state to 'enrolled'.

Note: The above Auto-Enroll-Response process is similar to the Trip-to-Pair process (described in *Section 37.3.2*) except user authorization for the enrollment of the Zone device is not required and no Zone Enroll Request command needs to be sent to the CIE device.

Parent topic: Enrollment

Auto-Enroll-Request The Auto-Enroll-Request method of enrollment is described below:

1. After the IAS Zone device joins the network, the CIE device performs a service discovery.
2. If the CIE device determines that it wants to enroll the Zone device, it sends a Write Attribute command to the Zone device in order to write its IEEE/MAC address to the relevant attribute.
3. The Zone device may optionally create a binding table entry for the CIE device and store the CIE device's IEEE/MAC address there.
4. The Zone device sends a Zone Enroll Request command to the CIE device.
5. The CIE device assigns a Zone ID to the Zone device and sends a Zone Enroll Response command to it.
6. The Zone device updates its attributes to stored the assigned Zone ID and update its zone state to 'enrolled'.

Note: The above Auto-Enroll-Request process is similar to the Trip-to-Pair process (described in Section 37.3.2) except that user authorization for the enrollment of the Zone device is not required.

Parent topic: Enrollment

Parent topic:*IAS Zone Cluster*

IAS Zone Events The IAS Zone cluster has its own events that are handled through the callback mechanism outlined in *Chapter 3*. If a device uses the IAS Zone cluster then IAS Zone event handling must be included in the callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function (for example, through **eHA_RegisterIASZoneEndPoint()** for a Zone device). The relevant callback function is then invoked when an IAS Zone event occurs.

For an IAS Zone event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_IASZoneCallBackMessage structure:

```
typedef struct
{
uint8 u8CommandId;
union
    {
     tsCLD_IASZone_TestModeUpdate                    *psTestModeUpdate; /* Internal */
     tsCLD_IASZone_EnrollRequestCallBackPayload      sZoneEnrollRequestCallbackPayload;
     tsCLD_IASZone_EnrollResponsePayload             *psZoneEnrollResponsePayload;
     tsCLD_IASZone_StatusChangeNotificationPayload   *psZoneStatusNotificationPayload;
     tsCLD_IASZone_InitiateTestModeRequestPayload    *psZoneInitiateTestModeRequestPayload;
    } uMessage;
} tsCLD_IASZoneCallBackMessage;
```

When an IAS Zone event occurs, one of several command types could have been received. The relevant command type is specified through the u8CommandId field of the tsSM_CallBackMessage structure. The possible command/event types are detailed in the table below (note that psTestModeUpdate is for internal use only).

u8CommandId Enumeration	Description
E_CLD_IASZONE_CMD_ZONE_ENI	An IAS Zone Enroll Response has been received by the cluster server
E_CLD_IASZONE_CMD_ZONE_STA	An IAS Zone Status Change Notification has been received by the cluster client
E_CLD_IASZONE_CMD_ZONE_ENI	An IAS Zone Enroll Request has been received by the cluster client
E_CLD_IASZONE_CMD_INITIATE_	An IAS Zone Normal Operation Mode Initiation Request command has been received by the cluster server
E_CLD_IASZONE_CMD_INITIATE_	An IAS Zone Initiate Test Mode Request has been received by the cluster server

Parent topic:*IAS Zone Cluster*

Functions The following IAS Zone cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_IASZoneCreateIASZone
2. eCLD_IASZoneUpdateZoneStatus
3. eCLD_IASZoneUpdateZoneState
4. eCLD_IASZoneUpdateZoneType
5. eCLD_IASZoneUpdateZoneID
6. eCLD_IASZoneUpdateCIEAddress
7. eCLD_IASZoneEnrollReqSend
8. eCLD_IASZoneEnrollRespSend

9. eCLD_IASZoneStatusChangeNotificationSend
10. eCLD_IASZoneNormalOperationModeReqSend
11. eCLD_IASZoneTestModeReqSend

eCLD_IASZoneCreateIASZone

```
teZCL_Status eCLD_IASZoneCreateIASZone(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits,
    tsCLD_IASZone_CustomDataStructure *psCustomDataStructure);
```

Description This function creates an instance of the IAS Zone cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create an IAS Zone cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* : Type of cluster instance (server or client) to be created: *TRUE - server FALSE - client*
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the IAS Zone cluster. This parameter can refer to a pre-filled structure called sCLD_``IASZone which is provided in the **IASZone.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_IASZone which defines the attributes of IAS Zone cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster.
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 37.6.1)

Returns

- E_ZCL_SUCCESS

- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_IASZoneUpdateZoneStatus

```
teZCL_Status eCLD_IASZoneUpdateZoneStatus(
    uint8 u8SourceEndPoint,
    uint16 u16StatusBitMask,
    bool_t bStatusState);
```

Description This function can be used on an IAS Zone cluster server to update the zone status bitmap stored in the b16ZoneStatus attribute, described in Section 37.2.

In one call to this function, one or more selected bits in the b16ZoneStatus attribute bitmap can be to set to '1' or '0'. The affected bits must themselves be specified in a bitmap and the value to be set must also be specified.

If the server is enrolled with a client on a CIE device, the function sends a notification of this update to the client, in a Zone Status Change Notification. Before sending the notification and returning, the function invokes a user-defined callback function to allow the application to validate the status change.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS Zone cluster resides
- *u16StatusBitMask*: 16-bit bitmap indicating the bits of the zb16ZoneStatus bitmap to be updated. There is a one-to-one correspondence between the bits of the two bitmaps and a bit should be set to '1' if the corresponding attribute bit is to be updated. Enumerations are provided (which can be logical-ORed):

Bits	Enumeration
0	CLD_IASZONE_STATUS_MASK_ALARM1
1	CLD_IASZONE_STATUS_MASK_ALARM2
2	CLD_IASZONE_STATUS_MASK_TAMPER
3	CLD_IASZONE_STATUS_MASK_BATTERY
4	CLD_IASZONE_STATUS_MASK_SUPERVISION_REPORTS
5	CLD_IASZONE_STATUS_MASK_RESTORE_REPORTS
6	CLD_IASZONE_STATUS_MASK_TROUBLE
7	CLD_IASZONE_STATUS_MASK_AC_MAINS
8	CLD_IASZONE_STATUS_MASK_TEST
9	CLD_IASZONE_STATUS_MASK_BATTERY_DEFECT
10-15	Reserved

- *bStatusState*: Boolean indicating the value to which the attribute bits to be updated must be set - enumerations are provided:
  - CLD_IASZONE_STATUS_MASK_SET (1)
  - CLD_IASZONE_STATUS_MASK_RESET (0)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Functions

eCLD_IASZoneUpdateZoneState

```
teZCL_Status eCLD_IASZoneUpdateZoneState(
    uint8 u8SourceEndPoint,
    teCLD_IASZoneState eZoneState);
```

Description This function can be used on an IAS Zone cluster server to update the zone state value stored in the `e8ZoneState` attribute, described in Section 37.2. This attribute indicates whether or not the server is enrolled with a client on a CIE device. The function checks that the specified state is valid.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS Zone cluster resides
- *eZoneState*: Zone state value to be written to the attribute, one of:
  - E_CLD_IASZONE_STATE_NOT_ENROLLED (0x00)
  - E_CLD_IASZONE_STATE_ENROLLED (0x01)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Functions

eCLD_IASZoneUpdateZoneType

```
teZCL_Status eCLD_IASZoneUpdateZoneType(
    uint8 u8SourceEndPoint,
    teCLD_IASZoneType eIASZoneType);
```

Description This function can be used on an IAS Zone cluster server to update the zone type value stored in the `e16ZoneType` attribute. The possible values are listed in Section 37.2 and the function checks that the specified type is one of these values.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS Zone cluster resides *eIAS-ZoneType*: Zone type value to be written to the attribute (for the possible values, refer to Section 37.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Functions

eCLD_IASZoneUpdateZoneID

```
teZCL_Status eCLD_IASZoneUpdateZoneID(
    uint8 u8SourceEndPoint,
    uint8 u8IASZoneId);
```

Description This function can be used on an IAS Zone cluster server to update the zone ID value stored in the `u8ZoneId` attribute. This is an 8-bit user-defined identifier.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS Zone cluster resides
- *u8IASZoneId*: Zone ID value to be written to the attribute

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Functions

eCLD_IASZoneUpdateCIEAddress

```
teZCL_Status eCLD_IASZoneUpdateCIEAddress(
    uint8 u8SourceEndPoint,
    u64IEEEAddress u64CIEAddress);
```

Description This function can be used on an IAS Zone cluster server to update the 64-bit IEEE/MAC address stored in the `u64IASCIEAddress` attribute. This is the address of the CIE device to which the local device should send commands and notifications.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS Zone cluster resides
- *u64CIEAddress*: IEEE/MAC address to be written to the attribute

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Functions

eCLD_IASZoneEnrollReqSend

```
teZCL_Status eCLD_IASZoneEnrollReqSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_IASZone_EnrollRequestPayload *psPayload);
```

Description This function can be used on an IAS Zone cluster server to send an IAS Zone Enroll Request to an IAS Zone client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: : Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 37.6.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASZoneEnrollRespSend

```
teZCL_Status eCLD_IASZoneEnrollRespSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_IASZone_EnrollResponsePayload *psPayload);
```

Description This function can be used on an IAS Zone cluster client to send an IAS Zone Enroll Response to the IAS Zone server.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to a structure containing the payload for the command (see Section 37.6.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASZoneStatusChangeNotificationSend

```
teZCL_Status eCLD_IASZoneStatusChangeNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
```

(continues on next page)

(continued from previous page)

```
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_IASZone_StatusChangeNotificationPayload
    *psPayload);
```

Description This function can be used on IAS Zone cluster server to send a Zone Status Change Notification to the IAS Zone client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 37.6.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASZoneNormalOperationModeReqSend

```
teZCL_Status eCLD_IASZoneNormalOperationModeReqSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on IAS Zone cluster client to send a request the IAS Zone server to initiate normal operation mode. If required, this command must be enabled in the compile-time options, as described in Section 37.7.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASZoneTestModeReqSend

```
teZCL_Status eCLD_IASZoneTestModeReqSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_IASZone_InitiateTestModeRequestPayload
    *psPayload);
```

Description This function can be used on IAS Zone cluster client to send a request to the IAS Zone server to initiate test mode and operate in this mode for a specified time. If required, this command must be enabled in the compile-time options, as described in Section 37.7.

Test mode allows the target device to be temporarily isolated from the IAS to allow configuration/adjustment of the device. Alternatively, the whole IAS can be put into test mode for maintenance, but the command issued by this function only affects the individual target IAS Zone cluster server(s).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 37.6.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

Parent topic:*IAS Zone Cluster*

Structures

Custom Data Structure The IAS Zone cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsCLD_IASZone_InitiateTestModeRequestPayload sTestMode;
    tsZCL_ReceiveEventAddress      sReceiveEventAddress;
    tsZCL_CallBackEvent            sCustomCallBackEvent;
```

(continues on next page)

(continued from previous page)

```
    tsCLD_IASZoneCallBackMessage   sCallBackMessage;
} tsCLD_IASZone_CustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Custom Command Payloads The following structures contain the payloads for the IAS Zone cluster custom commands.

'Enroll Request' Payload The following structure contains the payload of an Enroll Request command.

```
typedef struct
{
    zenum16     e16ZoneType;
    uint16      u16ManufacturerCode;
}tsCLD_IASZone_EnrollRequestPayload;
```

where:

- e16ZoneType is the zone type of the local (sending) node, as specified in the e``16ZoneType attribute (see Section 37.2)
- u16ManufacturerCode is the manufacturer ID code that is held in the Node Descriptor of the local (sending) node

'Enroll Response' Payload The following structure contains the payload of an Enroll Response command.

```
typedef struct
{
   teCLD_IASZoneZoneEnrollRspCode    e8EnrollResponseCode;
   uint8                             u8ZoneID;
}tsCLD_IASZone_EnrollResponsePayload;
```

where:

- e8EnrollResponseCode is a code indicating the outcome of the corresponding Enroll Request, one of:

Enumeration	Description
E_CLD_IASZONE_ENROLL_RESP_SUCCE	Requested enrollment successful
E_CLD_IASZONE_ENROLL_RESP_NOT_S	Zone type of requesting device is not known/supported by the CIE device
E_CLD_IASZONE_ENROLL_RESP_NO_EN	CIE device is not allowing new zones to be enrolled at the present time
E_CLD_IASZONE_ENROLL_RESP_TOO_M	CIE device has reached its limit for the number of zones that it can enroll

- u8ZoneID is the index of the entry for the enrollment which has been added to the Zone table on the CIE device (only valid for a successful enrollment)

'Zone Status Change Notification' Payload The following structure contains the payload of a Zone Status Change Notification command.

```
typedef struct
{
  zbmap16    b16ZoneStatus;
  zbmap8     b8ExtendedStatus;
  zuint8     u8ZoneId;
  zuint16    u16Delay;
}tsCLD_IASZone_StatusChangeNotificationPayload;
```

where:

- b16ZoneStatus contains the new/current status of the (sending) zone device, as indicated in the e``8ZoneState attribute - one of:
  - E_CLD_IASZONE_STATE_NOT_ENROLLED (0x01)
  - E_CLD_IASZONE_STATE_ENROLLED (0x02)
- b8ExtendedStatus can be optionally used to indicate further status information, but otherwise should be set to zero
- u8ZoneId is the index of the entry for the (sending) device in the Zone table on the CIE device
- u16Delay is is the time-delay, in quarter-seconds, between the status change taking place in the e``8ZoneState attribute and the successful transmission of the Zone Status Change Notification (this value can be used in assessing network traffic congestion)

'Initiate Test Mode Request' Payload The following structure contains the payload of an Initiate Test Mode Request command.

```
typedef struct
{
   uint8    u8TestModeDuration;
   uint8    u8CurrentZoneSensitivityLevel;
}tsCLD_IASZone_InitiateTestModeRequestPayload;
```

where:

- u8TestModeDuration is the duration, in seconds, for which the device should remain in test mode
- u8CurrentZoneSensitivityLevel is the current sensitivity level for the zone, as indicated in the u8CurrentZoneSensitivityLevel attribute (see Section 37.2)

Parent topic: Structures

Parent topic:*IAS Zone Cluster*

Compile-time options To enable the IAS Zone cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_IASZONE
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one of the following to the same file:

```
#define IASZONE_SERVER
#define IASZONE_CLIENT
```

Optional Attributes Add this line to enable the optional Number Of Zone Sensitivity Levels attribute:

```
#define CLD_IASZONE_ATTR_ID_NUMBER_OF_ZONE_SENSITIVITY_LEVELS
```

Add this line to enable the optional Current Zone Sensitivity Level attribute:

```
#define CLD_IASZONE_ATTR_ID_CURRENT_ZONE_SENSITIVITY_LEVEL
```

Global Attributes Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_IASZONE_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Optional Commands Add this line to enable the optional Initiate Normal Operation Mode command:

```
#define CLD_IASZONE_CMD_INITIATE_NORMAL_OPERATION_MODE
```

Add this line to enable the optional Initiate Test Mode command:

```
#define CLD_IASZONE_CMD_INITIATE_TEST_MODE
```

Disable APS Acknowledgements for Bound Transmissions APS acknowledgements for bound transmissions from this cluster can be disabled by defining:

```
#define CLD_IASZONE_BOUND_TX_WITH_APS_ACK_DISABLED
```

Parent topic:*IAS Zone Cluster*

IAS Ancillary Control Equipment Cluster This chapter describes the IAS Ancillary Control Equipment (ACE) cluster, which provides a control interface to a CIE (Control and Indicating Equipment) device in an IAS (Intruder Alarm System).

The IAS ACE cluster has a Cluster ID of 0x0501.

Overview The IAS ACE cluster provides a control interface to a CIE (Control and Indicating Equipment) device in an IAS (Intruder Alarm System). For example, it allows a remote control unit to be used to configure the IAS via a CIE device. The server side of the cluster is implemented on the CIE device and the client side is implemented on the remote device.

To use the functionality of this cluster, you must include the file **IASACE.h** in your application and enable the cluster by defining CLD_IASACE in the **zcl_options.h** file.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the IAS ACE cluster are fully detailed in Section 38.9.

Parent topic:*IAS Ancillary Control Equipment Cluster*

IAS ACE Structure and Attributes The structure definition for the IAS ACE cluster is shown below.

```
typedef struct
{
  zuint16      u16ClusterRevision;
}   tsCLD_IASACE;
```

where u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*IAS Ancillary Control Equipment Cluster*

Table and Parameters The IAS ACE cluster server hosts the following table and sets of parameters:

- **Zone table:** The Zone table contains an entry for each enrolled zone. Each entry stores the identifier and type of the zone, as well as the IEEE/MAC address of the device which hosts the zone (see Section 38.7.2).
- **Zone parameters:** This set of parameters contains certain zone properties including the zone status, the zone name/label and the zone arm/disarm code (see Section 38.7.3)
- **Panel parameters:** This set of parameters contains certain status information about the display panel and alarm (see Section 38.7.4).

Parent topic:*IAS Ancillary Control Equipment Cluster*

Command Summary The IAS ACE cluster includes a number of commands that can be sent by the application on the client or server. These commands are summarised below.

- *Table 50* lists the commands that can be issued on the client.
- *Table 51* lists the commands that can be issued on the server.

Functions are provided to send these commands - these functions are indicated in the descriptions below and detailed in Section 38.6.

IAS ACE Cluster Commands from Client to Server

Command	Description and Function
Arm	Instructs the server to put all or certain enrolled zones into the 'armed' state or put all of them into the 'disarmed' state.eCLD_IASACE_ArmSend()
Bypass	Instructs the server to take one or more specified zones out of the system for the current activation (these zones are reinstated the next time the system is dis-armed and to exclude them again the next time the system is armed, the Bypass command must be re-sent before sending the Arm command).eCLD_IASACE_BypassSend()
Emergency	Instructs the server to put the alarm in the 'Emergency' state.eCLD_IASACE_EmergencySend()
Fire	Instructs the server to put the alarm in the 'Fire' state.eCLD_IASACE_FireSend()
Panic	Instructs the server to put the alarm in the 'Panic' state.eCLD_IASACE_PanicSend()
Get Zone ID Map	Requests the Zone IDs that have been allocated to zones.eCLD_IASACE_GetZoneIDMapSend()
Get Zone Information	Requests information on a specified zone.eCLD_IASACE_GetZoneInfoSend()
Get Panel Status	Requests the current status of the (display) panel.eCLD_IASACE_GetPanelStatusSend()
Get Bypassed Zone List	Requests a list of the currently bypassed zones. eCLD_IASACE_GetBypassedZoneListSend()
Get Zone Status	Requests a list of either all zones with their status or those zones with a particular status (that is, all zones with the b16ZoneStatus attribute of the IAS Zone cluster having a certain value). eCLD_IASACE_GetZoneStatusSend()

IAS ACE Cluster Commands from Server to Client

Command	Description and Function
Set Bypassed Zone List	Informs the client which zones are currently bypassed and can be sent in response to a Bypassed Zone List command, Get eCLD_IASACE_SetBypassedZoneListSend()
Zone Status Changed	Informs the client that the status (value of the b16ZoneStatus attribute of the IAS Zone cluster) of a particular zone has changed.eCLD_IASACE_ZoneStatusChangedSend()
Panel Status Changed	Informs the client that the status of the (display) panel has changed. eCLD_IASACE_PanelStatusChanged()

Parent topic:*IAS Ancillary Control Equipment Cluster*

IAS ACE Events The IAS ACE cluster has its own events that are handled through the callback mechanism outlined in *Chapter 3*. If a device uses the IAS ACE cluster then IAS ACE event handling must be included in the callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function (for example, through

eHA_RegisterIASCIEEndPoint() for a CIE device). The relevant callback function will then be invoked when an IAS ACE event occurs.

For an IAS ACE event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_IASACECallBackMessage structure:

```
typedef struct
{
uint8    u8CommandId;
  union
  {
  tsCLD_IASACE_ArmPayload  *psArmPayload;
  tsCLD_IASACE_BypassPayload     *psBypassPayload;
  tsCLD_IASACE_GetZoneInfoPayload   *psGetZoneInfoPayload;
  tsCLD_IASACE_GetZoneStatusPayload   *psGetZoneStatusPayload;
  tsCLD_IASACE_ArmRespPayload      *psArmRespPayload;
  tsCLD_IASACE_GetZoneIDMapRespPayload   *psGetZoneIDMapRespPayload;
  tsCLD_IASACE_GetZoneInfoRespPayload   *psGetZoneInfoRespPayload;
  tsCLD_IASACE_ZoneStatusChangedPayload   *psZoneStatusChangedPayload;
  tsCLD_IASACE_PanelStatusChangedOrGetPanelStatusRespPayload
   *psPanelStatusChangedOrGetPanelStatusRespPayload;
  tsCLD_IASACE_SetBypassedZoneListPayload    *psSetBypassedZoneListPayload;
  tsCLD_IASACE_BypassRespPayload    *psBypassRespPayload;
  tsCLD_IASACE_GetZoneStatusRespPayload   *psGetZoneStatusRespPayload;
    } uMessage;
} tsCLD_IASACECallBackMessage;
```

When an IAS ACE event occurs, one of twelve command types could have been received. The relevant command type is specified through the u8CommandId field of the tsCLD_IASACECallBackMessage structure. The possible command/event types are detailed in *Table 52* below (for command descriptions, refer to Section 38.4).

In the case where an IAS Arm or Bypass command has been received and results in a change to a Zone parameter on the cluster server (e.g. an update of the zone status u8ZoneStatusFlag), a second event will be generated before any response is sent. This is a 'cluster update' event for which the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_UPDATE. This prompts the application to perform any required actions such as saving persistent data and refreshing a display.

u8CommandId Enumeration	Description
Server Events	
E_CLD_IASACE_CMD_ARM	An IAS ACE Arm command has been received by the server
E_CLD_IASACE_CMD_BYPASS	An IAS ACE Bypass command has been received by the server
E_CLD_IASACE_CMD_EMERGENCY	An IAS ACE Emergency command has been received by the server
E_CLD_IASACE_CMD_FIRE	An IAS ACE Fire command has been received by the server
E_CLD_IASACE_CMD_PANIC	An IAS ACE Panic command has been received by the server
E_CLD_IASACE_CMD_GET_ZONE_I	An IAS ACE Get Zone ID Map command has been received by the server
E_CLD_IASACE_CMD_GET_ZONE_I	An IAS ACE Get Zone Information command has been received by the server
E_CLD_IASACE_CMD_GET_PANEL_	An IAS ACE Get Panel Status command has been received by the server
E_CLD_IASACE_CMD_GET_BYPASS	An IAS ACE Get Bypassed Zone List command has been received by the server
E_CLD_IASACE_CMD_GET_ZONE_S	An IAS ACE Get Zone Status command has been received by the server
Client Events	
E_CLD_IASACE_CMD_ARM_RESP	An IAS ACE Arm Response command has been received by the client
E_CLD_IASACE_CMD_GET_ZONE_I	An IAS ACE Get Zone ID Map Response command has been received by the client
E_CLD_IASACE_CMD_GET_ZONE_I	An IAS ACE Get Zone Information Response command has been received by the client
E_CLD_IASACE_CMD_ZONE_STATU	An IAS ACE Zone Status Changed command has been received by the client
E_CLD_IASACE_CMD_PANEL_STAT	An IAS ACE Panel Status Changed command has been received by the client
E_CLD_IASACE_CMD_GET_PANEL_	An IAS ACE Get Panel Status Response command has been received by the client
E_CLD_IASACE_CMD_SET_BYPASSED_ZONE_LIST	An IAS ACE Set Bypassed Zone List command has been received by the client
E_CLD_IASACE_CMD_BYPASS_RES	An IAS ACE Bypass Response command has been received by the client
E_CLD_IASACE_CMD_GET_ZONE_S	An IAS ACE Get Zone Status Response command has been received by the client

Parent topic:*IAS Ancillary Control Equipment Cluster*

Functions The following IAS ACE cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_IASACECreateIASACE
2. eCLD_IASACEAddZoneEntry
3. eCLD_IASACERemoveZoneEntry
4. eCLD_IASACEGetZoneTableEntry
5. eCLD_IASACEGetEnrolledZones

6. eCLD_IASACESetPanelParameter
7. eCLD_IASACEGetPanelParameter
8. eCLD_IASACESetZoneParameter
9. eCLD_IASACESetZoneParameterValue
10. eCLD_IASACEGetZoneParameter
11. eCLD_IASACE_ArmSend
12. eCLD_IASACE_BypassSend
13. eCLD_IASACE_EmergencySend
14. eCLD_IASACE_FireSend
15. eCLD_IASACE_PanicSend
16. eCLD_IASACE_GetZoneIDMapSend
17. eCLD_IASACE_GetZoneInfoSend
18. eCLD_IASACE_GetPanelStatusSend
19. eCLD_IASACE_SetBypassedZoneListSend
20. eCLD_IASACE_GetBypassedZoneListSend
21. eCLD_IASACE_GetZoneStatusSend
22. eCLD_IASACE_ZoneStatusChangedSend
23. eCLD_IASACE_PanelStatusChanged

eCLD_IASACECreateIASACE

```
teZCL_Status eCLD_IASACECreateIASACE(
tsZCL_ClusterInstance *psClusterInstance,
bool_t bIsServer,
tsZCL_ClusterDefinition *psClusterDefinition,
void *pvEndPointSharedStructPtr,
tsCLD_IASACECustomDataStructure
*psCustomDataStructure);
```

Description This function creates an instance of the IAS ACE cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create an IAS ACE cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.

- *bIsServer*: Type of cluster instance (server or client) to be created: *TRUE - server*, *FALSE - client*
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the IAS ACE cluster.
- *pvEndPointSharedStructPtr*: Set this pointer to NULL for this cluster
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 38.7.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_IASACEAddZoneEntry

```
teZCL_CommandStatus eCLD_IASACEAddZoneEntry(
    uint8 u8SourceEndPointId,
    uint16 u16ZoneType,
    uint64 u64IeeeAddress,
    uint8 *pu8ZoneID);
```

Description This function can be used on an IAS ACE cluster server to create an entry in the local Zone table - that is, to add the details of a zone to the table after receiving a Zone Enrollment Request (and before sending a Zone Enrollment Response).

The details of the zone are provided in the function parameters. The function checks that the supplied pointer to the Zone ID is not NULL and that the supplied IEEE address is valid. The function can then add the zone details to the Zone table, provided that there is a free entry in the table.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *u16ZoneType*: Value indicating the type of zone to be added to the table (for the possible values, refer to the description of the attribute e``16ZoneType of the IAS Zone cluster in Section 37.2)
- *u64IeeeAddress*: IEEE address of the device which hosts the zone
- *pu8ZoneID*: Pointer to an identifier of the zone to be added to the table

Returns

- E_ZCL_CMDS_SUCCESS (zone successfully added to Zone table)
- E_ZCL_CMDS_FAILURE (cluster instance not found)
- E_ZCL_CMDS_INVALID_FIELD (pointer to Zone ID is NULL)
- E_ZCL_CMDS_INVALID_VALUE (IEEE address is invalid)

- E_ZCL_CMDS_INSUFFICIENT_SPACE (no free entry in Zone table)

Parent topic: Functions

eCLD_IASACERemoveZoneEntry

```
teZCL_CommandStatus eCLD_IASACERemoveZoneEntry(
  uint8 u8SourceEndPointId,
    uint8 u8ZoneID,
    uint64 *pu64IeeeAddress);
```

Description This function can be used on an IAS ACE cluster server to remove an existing entry from the local Zone table - that is, to delete the details of a zone in the table and release the table entry for re-use. Thus, this function can be used to unenroll a zone.

The zone to be removed is specified by means of the Zone ID. The function checks that the supplied pointer to a location to receive the IEEE address is not NULL. The function then searches for the relevant table entry using the supplied Zone ID and, if found, returns its IEEE address via the supplied location and frees the table entry by setting the IEEE address in the table entry to zero. The returned IEEE address can be used by a (local) CIE device application to send a request to the relevant Zone device to set its IAS Zone cluster attribute `u64IASCIEAddress` to all zeros (writing to remote attributes is described in Section 2.3.3.1).

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *u8ZoneID*: Zone ID of zone to be removed from table
- *pu64IeeeAddress*: Pointer to location to receive the IEEE address found in the table entry to be removed

Returns

- E_ZCL_CMDS_SUCCESS (zone successfully removed from Zone table)
- E_ZCL_CMDS_FAILURE (cluster instance not found)
- E_ZCL_CMDS_INVALID_FIELD (pointer to IEEE address location is NULL)
- E_ZCL_CMDS_NOT_FOUND (entry with specified Zone ID not found in table)

Parent topic: Functions

eCLD_IASACEGetZoneTableEntry

```
teZCL_CommandStatus eCLD_IASACEGetZoneTableEntry(
    uint8 u8SourceEndPointId,
    uint8 u8ZoneID,
    tsCLD_IASACE_ZoneTable **ppsZoneTable);
```

Description This function can be used on an IAS ACE cluster server to obtain the details of a specified zone from the local Zone table.

The zone of interest is specified by means of its Zone ID. The function searches for the relevant table entry using the supplied Zone ID and, if found, returns the zone information from the table entry via the supplied structure (see Section 38.7.2).

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *u8ZoneID*: Zone ID of zone for which details required from table
- *ppsZoneTable*: Pointer to a pointer to a structure to receive obtained zone information (see Section 38.7.2)

Returns

- E_ZCL_CMDS_SUCCESS (zone details successfully obtained from Zone table)
- E_ZCL_CMDS_FAILURE (cluster instance not found)
- E_ZCL_CMDS_NOT_FOUND (entry with specified Zone ID not found in table)

Parent topic: Functions

eCLD_IASACEGetEnrolledZones

```
teZCL_CommandStatus eCLD_IASACEGetEnrolledZones(
    uint8 u8SourceEndPointId,
    uint8 *pu8ZoneID,
    uint8 *pu8NumOfEnrolledZones);
```

Description This function can be used on an IAS ACE cluster server to obtain a list of the enrolled zones from the local Zone table.

The function searches the Zone table and returns a list of the Zone IDs of all the enrolled zones (for which there are table entries). The number of enrolled zones is also returned.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *pu8ZoneID*: Pointer to a location to receive the first Zone ID in the reported list of enrolled zones
- *pu8NumOfEnrolledZones*: Pointer to a location to receive the number of enrolled zones reported in the above list

Returns

- E_ZCL_CMDS_SUCCESS (zone list successfully obtained from Zone table)
- E_ZCL_CMDS_FAILURE (cluster instance not found)
- E_ZCL_CMDS_INVALID_FIELD (a supplied pointer is NULL)

Parent topic: Functions

eCLD_IASACESetPanelParameter

```
teZCL_Status eCLD_IASACESetPanelParameter(
    uint8 u8SourceEndPointId,
    teCLD_IASACE_PanelParameterID eParameterId,
    uint8 u8ParameterValue);
```

Description This function can be used on an IAS ACE cluster server to set the value of a Panel parameter. The Panel parameters are held on the server in a tsCLD_IASACE_PanelParameter structure (see Section 38.7.4) and this function can be used to write a value to one parameter in the structure. The function verifies that the specified parameter identifier is valid before attempting the write.

If this function is used to set the Panel parameter ePanelStatus, an IAS ACE Panel Status Changed command is automatically sent to all bound clients.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *eParameterId*: Enumeration identifying the Panel parameter to be set, one of:
  - E_CLD_IASACE_PANEL_PARAMETER_PANEL_STATUS
  - E_CLD_IASACE_PANEL_PARAMETER_SECONDS_REMAINING
  - E_CLD_IASACE_PANEL_PARAMETER_AUDIBLE_NOTIFICATION
  - E_CLD_IASACE_PANEL_PARAMETER_ALARM_STATUS
- *u8ParameterValue*: Value to be written to the parameter

Returns

- E_ZCL_SUCCESS (Panel parameter successfully set)
- E_ZCL_ERR_CLUSTER_NOT_FOUND (cluster instance not found)
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND (Panel parameter identifier invalid)

Parent topic: Functions

eCLD_IASACEGetPanelParameter

```
teZCL_Status eCLD_IASACEGetPanelParameter(
    uint8 u8SourceEndPointId,
    teCLD_IASACE_PanelParameterID eParameterId,
    uint8 *pu8ParameterValue);
```

Description This function can be used on an IAS ACE cluster server to obtain the value of a Panel parameter. The Panel parameters are held on the server in a tsCLD_IASACE_PanelParameter structure (see Section 38.7.4) and this function can be used to read the value of one parameter in the structure. The function verifies that the specified parameter identifier is valid before attempting the read.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *eParameterId* : Enumeration identifying the Panel parameter to be read, one of:
- E_CLD_IASACE_PANEL_PARAMETER_PANEL_STATUS
- E_CLD_IASACE_PANEL_PARAMETER_SECONDS_REMAINING
- E_CLD_IASACE_PANEL_PARAMETER_AUDIBLE_NOTIFICATION
- E_CLD_IASACE_PANEL_PARAMETER_ALARM_STATUS
- *pu8ParameterValue*: Pointer to location to receive read parameter value

Returns

- E_ZCL_SUCCESS (Panel parameter successfully read)
- E_ZCL_ERR_CLUSTER_NOT_FOUND (cluster instance not found)
- E_ZCL_ERR_PARAMETER_NULL (specfied pointer is NULL)
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND (Panel parameter identifier invalid)

Parent topic: Functions

eCLD_IASACESetZoneParameter

```
teZCL_Status eCLD_IASACESetZoneParameter(
    uint8 u8SourceEndPointId,
    teCLD_IASACE_ZoneParameterID eParameterId,
    uint8 u8ZoneID,
    uint8 u8ParameterLength,
    uint8 *pu8ParameterValue);
```

Description This function can be used on an IAS ACE cluster server to set the value of a Zone parameter. The Zone parameters for a particular Zone ID are held on the server in a tsCLD_IASACE_ZoneParameter structure (see Section 38.7.3) and this function can be used to write a value to one parameter in the structure. The specified zone must have been enrolled in the local Zone table. Before attempting the write, the function verifies that the specified Zone ID is present in the Zone table and that the specified parameter identifier is valid.

If this function is used to set the Zone parameter eZoneStatus, an IAS ACE Zone Status Changed command is automatically sent to all bound clients.

The function requires the parameter value to be provided as a **uint8** array. This is to allow one of the array parameters, au8ZoneLabel[] or au8ArmDisarmCode[], to be set - the corresponding string parameter, sZoneLabel or sArmDisarmCode, will be set automatically. The function **eCLD_IASACESetZoneParameterValue()** provides an easier way of setting one of the non-array/non-string parameters.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *eParameterId* : Enumeration identifying the Zone parameter to be set, one of:
- : E_CLD_IASACE_ZONE_PARAMETER_ZONE_CONFIG_FLAG
- E_CLD_IASACE_ZONE_PARAMETER_ZONE_STATUS_FLAG
- E_CLD_IASACE_ZONE_PARAMETER_ZONE_STATUS
- E_CLD_IASACE_ZONE_PARAMETER_AUDIBLE_NOTIFICATION
- E_CLD_IASACE_ZONE_PARAMETER_ZONE_LABEL
- E_CLD_IASACE_ZONE_PARAMETER_ARM_DISARM_CODE
- *u8ZoneID*: Zone ID of zone information to be updated
- *u8ParameterLength*: Number of *uint8* elements in the array containing the parameter value to be set
- *pu8ParameterValue*: Pointer to a location containing the first element of the array containing the parameter value to be set

Returns

- E_ZCL_SUCCESS (Zone parameter successfully set)
- E_ZCL_ERR_CLUSTER_NOT_FOUND (cluster instance not found)
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND (Zone parameter identifier invalid)
- E_ZCL_ERR_NO_REPORT_ENTRIES (Zone ID not found in Zone table)
- E_ZCL_ERR_PARAMETER_NULL (Pointer to location containing value is NULL)
- E_ZCL_ERR_PARAMETER_RANGE (specified array length too long to be stored)

Parent topic: Functions

eCLD_IASACESetZoneParameterValue

```
teZCL_Status eCLD_IASACESetZoneParameterValue(
    uint8 u8SourceEndPointId,
    teCLD_IASACE_ZoneParameterID eParameterId,
    uint8 u8ZoneID,
    uint16 u16ParameterValue);
```

Description This function can be used on an IAS ACE cluster server to set the value of a Zone parameter. The Zone parameters for a particular Zone ID are held on the server in a tsCLD_IASACE_ZoneParameter structure (see Section 38.7.3) and this function can be used to write a value to one of the non-string/non-array parameters in the structure. The specified zone must have been enrolled in the local Zone table. Before attempting the write, the function verifies that the specified Zone ID is present in the Zone table and that the specified parameter identifier is valid.

If this function is used to set the Zone parameter eZoneStatus, an IAS ACE Zone Status Changed command is automatically sent to all bound clients.

This function cannot be used to set the string parameters sZoneLabel and sArmDisarmCode or the array parameters au8ZoneLabel[] and au8ArmDisarmCode[]. The function **eCLD_IASACESetZoneParameter()** must be used to set the string and array parameters.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *eParameterId* : Enumeration identifying the Zone parameter to be set, one of:
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_CONFIG_FLAG
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_STATUS_FLAG
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_STATUS
  - E_CLD_IASACE_ZONE_PARAMETER_AUDIBLE_NOTIFICATION
- *u8ZoneID*: Zone ID of zone information to be updated
- *u16ParameterValue*: Value to be written to the parameter

Returns

- E_ZCL_SUCCESS (Zone parameter successfully set)
- E_ZCL_ERR_CLUSTER_NOT_FOUND (cluster instance not found)
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND (Zone parameter identifier invalid)

- E_ZCL_ERR_NO_REPORT_ENTRIES (Zone ID not found in Zone table)

Parent topic: Functions

eCLD_IASACEGetZoneParameter

```
teZCL_Status eCLD_IASACEGetZoneParameter(
    uint8 u8SourceEndPointId,
    teCLD_IASACE_ZoneParameterID eParameterId,
    uint8 u8ZoneID,
    uint8 *pu8ParameterLength,
    uint8 *pu8ParameterValue);
```

Description This function can be used on an IAS ACE cluster server to obtain the value of a Zone parameter. The Zone parameters for a particular Zone ID are held on the server in a tsCLD_IASACE_ZoneParameter structure (see Section 38.7.3) and this function can be used to read the value of one parameter in the structure. Before attempting the read, the function verifies that the specified Zone ID is present in the Zone table and that the specified parameter identifier is valid.

The function expects the read parameter value to be returned as a **uint8** array.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *eParameterId*: Enumeration identifying the Zone parameter to be read, one of:
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_CONFIG_FLAG
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_STATUS_FLAG
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_STATUS
  - E_CLD_IASACE_ZONE_PARAMETER_AUDIBLE_NOTIFICATION
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_LABEL
  - E_CLD_IASACE_ZONE_PARAMETER_ARM_DISARM_CODE
- *u8ZoneID*: Zone ID of zone information to be accessed
- **pu8ParameterLength*: Pointer to location to receive the number of *uint8* elements in the array containing the parameter value obtained
- **pu8ParameterValue*: Pointer to location to receive the first element of the array containing the parameter value obtained

Returns

- E_ZCL_SUCCESS (Zone parameter successfully read)
- E_ZCL_ERR_CLUSTER_NOT_FOUND (cluster instance not found)
- E_ZCL_ERR_PARAMETER_NULL (a specified pointer is NULL)
- E_ZCL_ERR_NO_REPORT_ENTRIES (Zone ID not found in Zone table)
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND (Zone parameter identifier invalid)
- E_ZCL_ERR_PARAMETER_RANGE (returned array too long to be stored)

Parent topic: Functions

eCLD_IASACE_ArmSend

```
teZCL_Status eCLD_IASACE_ArmSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_IASACE_ArmPayload *psPayload);
```

Description This function can be used on an IAS ACE cluster client to send an IAS ACE Arm command to an IAS ACE server. This command instructs the server to put all or certain enrolled zones into the 'armed' state or put all of them into the 'disarmed' state, according to the command payload (see Section 38.7.5).

The outcome of the request will be returned by the server in a response which will generate an E_CLD_IASACE_CMD_ARM_RESP event when received on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASACE_BypassSend

```
teZCL_Status eCLD_IASACE_BypassSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_IASACE_BypassPayload *psPayload);
```

Description This function can be used on an IAS ACE cluster client to send an IAS ACE Bypass command to an IAS ACE server. This command instructs the server to take one or more specified zones out of the system for the current activation.

Note: The bypassed zones will be reinstated the next time the system is disarmed. To exclude them again the next time the system is armed, the Bypass command must be re-sent before sending the Arm command.

The outcome of the request will be returned by the server in a response which will generate an E_CLD_IASACE_CMD_BYPASS_RESP event when received on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASACE_EmergencySend

```
teZCL_Status eCLD_IASACE_EmergencySend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on an IAS ACE cluster client to send an IAS ACE Emergency command to an IAS ACE server. This command instructs the server to put the alarm in the 'Emergency' state.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASACE_FireSend

```
teZCL_Status eCLD_IASACE_FireSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on an IAS ACE cluster client to send an IAS ACE Fire command to an IAS ACE server. This command instructs the server to put the alarm in the 'Fire' state.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASACE_PanicSend

```
teZCL_Status eCLD_IASACE_PanicSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on an IAS ACE cluster client to send an IAS ACE Panic command to an IAS ACE server. This command instructs the server to put the alarm in the 'Panic' state.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASACE_GetZoneIDMapSend

```
teZCL_Status eCLD_IASACE_GetZoneIDMapSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on an IAS ACE cluster client to send an IAS ACE Get Zone ID Map command to an IAS ACE server. This command requests the Zone IDs that have been allocated to zones.

The requested information is returned by the server in a response which generates an E_CLD_IASACE_CMD_GET_ZONE_ID_MAP_RESP event when received on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values

- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASACE_GetZoneInfoSend

```
teZCL_Status eCLD_IASACE_GetZoneInfoSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_IASACE_GetZoneInfoPayload *psPayload);
```

Description This function can be used on an IAS ACE cluster client to send an IAS ACE Get Zone Information command to an IAS ACE server. This command requests information on the zone specified in the command payload.

The requested information will be returned by the server in a response which will generate an E_CLD_IASACE_CMD_GET_ZONE_INFO_RESP event when received on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASACE_GetPanelStatusSend

```
teZCL_Status eCLD_IASACE_GetPanelStatusSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on an IAS ACE cluster client to send an IAS ACE Get Panel Status command to an IAS ACE server. This command requests the current status of the (display) panel.

The requested information will be returned by the server in a response which will generate an E_CLD_IASACE_CMD_GET_PANEL_STATUS_RESP event when received on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent

- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASACE_SetBypassedZoneListSend

```
teZCL_Status eCLD_IASACE_SetBypassedZoneListSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_IASACE_SetBypassedZonelistPayload *psPayload);
```

Description This function can be used on an IAS ACE cluster server to send an IAS ACE Set Bypassed Zone List command to an IAS ACE client. This command informs the client which zones are currently bypassed - the zones are specified in the command payload.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASACE_GetBypassedZoneListSend

```
teZCL_Status eCLD_IASACE_GetBypassedZoneListSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on an IAS ACE cluster client to send an IAS ACE Get Bypassed Zone List command to an IAS ACE server. This command requests a list of the currently bypassed zones.

The requested information will be returned by the server in a response which will generate an E_CLD_IASACE_CMD_SET_BYPASSED_ZONE_LIST event when received on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASACE_GetZoneStatusSend

```
teZCL_Status eCLD_IASACE_GetZoneStatusSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_IASACE_GetZoneStatusPayload *psPayload);
```

Description This function can be used on an IAS ACE cluster client to send an IAS ACE Get Zone Status command to an IAS ACE server. This command requests either of the following:

- a list of all enrolled zones with their status
- a list of those zones with a particular status (that is, all zones with the `b16ZoneStatus` attribute of the IAS Zone cluster having a certain value)

The list required is specified in the `bZoneStatusMaskFlag` field of the command payload (see Section 38.7.5). If the second of the above lists is required, the status to look for is also specified in the payload.

The requested information is returned by the server in a response which generates an E_CLD_IASACE_CMD_GET_ZONE_STATUS_RESP event when received on the client. A single response may not be able to carry all the zone status information to be returned and more than one request (and associated response) would be needed. For this reason, the request allows a starting zone and the number of zones to be included in the response to be specified (in the request payload).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASACE_ZoneStatusChangedSend

```
teZCL_Status eCLD_IASACE_ZoneStatusChangedSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_IASACE_ZoneStatusChangedPayload *psPayload);
```

Description This function can be used on an IAS ACE cluster server to send an IAS ACE Zone Status Changed command to an IAS ACE client. This command informs the client that the status of the specified zone has changed - that is, the value of the `b16ZoneStatus` attribute of the IAS Zone cluster for the zone has changed.

Note: This command is sent automatically when the function **eCLD_IASACESetZoneParameter()** is called on the server to update the `u16ZoneStatus` attribute for all the bound clients.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASACE_PanelStatusChanged

```
teZCL_Status eCLD_IASACE_PanelStatusChanged(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        teCLD_IASACE_ServerCmdId eCommandId,
        tsCLD_IASACE_PanelStatusChangedOrGetPanelStatusRespPayload
        *psPayload);
```

Description This function can be used on an IAS ACE cluster server to send an IAS ACE Panel Status Changed command to an IAS ACE client. This command informs the client that the value of the panel parameter `ePanelStatus` (see Section 38.7.4) on the (local) CIE device has changed.

Note:

1. The IAS ACE Panel Status Changed command is sent automatically when the function **eCLD_IASACESetPanelParameter()** is called to update the `ePanelStatus` parameter.
2. The function alternatively provides the option of sending an IAS ACE Get Panel Status Response but, in practice, this response is sent automatically when a Get Panel Status Request is received.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *eCommandId*: Identifier of command to be sent - for Panel Status Changed command, always set to:: E_CLD_IASACE_CMD_PANEL_STATUS_CHANGED
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

Parent topic:*IAS Ancillary Control Equipment Cluster*

Structures

Custom Data Structure The IAS ACE cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress          sReceiveEventAddress;
    tsZCL_CallBackEvent                sCustomCallBackEvent;
    tsCLD_IASACECallBackMessage        sCallBackMessage;
#if (defined CLD_IASACE) && (defined IASACE_SERVER)
    tsCLD_IASACE_PanelParameter
                                  sCLD_IASACE_PanelParameter;
    tsCLD_IASACE_ZoneParameter
          asCLD_IASACE_ZoneParameter[CLD_IASACE_ZONE_TABLE_SIZE];
    tsCLD_IASACE_ZoneTable
          asCLD_IASACE_ZoneTable[CLD_IASACE_ZONE_TABLE_SIZE];
#endif
} tsCLD_IASACECustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Zone Table Entry The following structure contains a Zone table entry, used to hold the enrollment details of a zone.

```
typedef struct
{
    zuint8              u8ZoneID;
    zbmap16             u16ZoneType;
    zieeeaddress        u64IeeeAddress;
} tsCLD_IASACE_ZoneTable;
```

where:

- u8ZoneID is the identifier of the zone
- u16ZoneType is a value indicating the type of zone (for the possible values, refer to the description of the attribute e``16ZoneType of the IAS Zone cluster in Section 37.2)
- u64IeeeAddress is the IEEE/MAC address of the device which hosts the zone

Parent topic: Structures

Zone Parameters The following structure is used to store the 'zone parameters' on the IAS ACE cluster server.

```
typedef struct
{
    zbmap8                      u8ZoneConfigFlag;
    zbmap8                      u8ZoneStatusFlag;
    zbmap16                     eZoneStatus;
    zenum8                      eAudibleNotification;
    tsZCL_CharacterString       sZoneLabel;
    uint8
        au8ZoneLabel[CLD_IASACE_MAX_LENGTH_ZONE_LABEL];
    tsZCL_CharacterString       sArmDisarmCode;
    uint8
        au8ArmDisarmCode[CLD_IASACE_MAX_LENGTH_ARM_DISARM_CODE];
}tsCLD_IASACE_ZoneParameter;
```

where:

- u8ZoneConfigFlag is is a bitmap used to configure the temporal role of a zone (as Day, Night or Day/Night) and whether the zone is allowed to be bypassed. Macros are provided as follows:

Bit	Macro
0	CLD_IASACE_ZONE_CONFIG_FLAG_BYPASS *
1	CLD_IASACE_ZONE_CONFIG_FLAG_DAY_HOME
2	CLD_IASACE_ZONE_CONFIG_FLAG_NIGHT_SLEEP
3	CLD_IASACE_ZONE_CONFIG_FLAG_NOT_BYPASSED **
4-7	Reserved

- * Determines whether the zone is allowed to be bypassed: 1 - allowed, 0 - not allowed
- ** Used to configure a status of ZONE_NOT_BYPASSED in responses to Bypass commands
- u8ZoneStatusFlag is a bitmap used to indicate the current status of a zone as armed or bypassed. Macros are provided as follows:

Bit	Macro
0	CLD_IASACE_ZONE_STATUS_FLAG_BYPASS
1	CLD_IASACE_ZONE_STATUS_FLAG_ARM
2-7	Reserved

- e``ZoneStatus is the zone status as the value of the b16ZoneStatus attribute of the IAS Zone cluster (see Section 37.2)
- eAudibleNotification is a value specifying whether an audible notification (e.g. a chime) is required to signal a zone status change (enumerations are available in teCLD_IASACE_AudibleNotification - see Section 38.8.4):

Value	Status
0x00	Audible notification muted
0x01	Audible notification sounded
0x02 - 0xFF	Reserved

- sZoneLabel is the name/label for the zone represented as a character string
- au8ZoneLabel[] is the name/label for the zone represented as an array of ASCII values
- sArmDisarmCode is the arm/disarm code for the zone represented as a character string
- au8ArmDisarmCode[] is the arm/disarm code for the zone represented as an array of ASCII values

Parent topic: Structures

Panel Parameters The following structure is used to store the 'panel parameters' on the IAS ACE cluster server.

```
typedef struct
{
    zenum8     ePanelStatus;
    zuint8     u8SecondsRemaining;
    zenum8     eAudibleNotification;
    zenum8     eAlarmStatus;
}tsCLD_IASACE_PanelParameter;
```

where:

- ePanelStatus is a value indicating the status to be displayed on the panel, as follows (enumerations are available in teCLD_IASACE_PanelStatus - see Section 38.8.2):

Value	Status
0x00	Disarmed (all zones) and ready to be armed
0x01	Armed stay
0x02	Armed night
0x03	Armed away
0x04	Exit delay
0x05	Entry delay
0x06	Not ready to be armed
0x07	In alarm
0x08	Arming stay
0x09	Arming night
0x0A	Arming away
0x0B - 0xFF	Reserved

- u8SecondsRemaining represents the time, in seconds, that the server will remain in the displayed state when the latter is 'Exit delay' or 'Entry delay' (for other states, this field should be set to 0x00).
- eAudibleNotification is a value specifying whether an audible notification (e.g. a chime) is required to signal a zone status change (enumerations are available in teCLD_IASACE_AudibleNotification - see Section 38.8.4):

Value	Status
0x00	Audible notification muted
0x01	Audible notification sounded
0x02 - 0xFF	Reserved

- eAlarmStatus is a value indicating the alarm status/type when the panel's state is 'In Alarm', as follows (enumerations are available in teCLD_IASACE_AlarmStatus - see Section 38.8.3):

Value	Status
0x00	No alarm
0x01	Burglar
0x02	Fire
0x03	Emergency
0x04	Police panic
0x05	Fire panic
0x06	Emergency panic
0x07 - 0xFF	Reserved

Parent topic: Structures

Custom Command Payloads The following structures contain the payloads for the IAS ACE cluster custom commands.

'Arm' Command Payload The following structure contains the payload of a Arm command.

```
typedef struct
{
    zenum8                  eArmMode;
    tsZCL_CharacterString   sArmDisarmCode;
    zuint8                  u8ZoneID;
} tsCLD_IASACE_ArmPayload;
```

where:

- eArmMode is a value indicating the state of armament in which to put the zone (enumerations are available in teCLD_IASACE_ArmMode - see Section38.8.1):

Value	Status
0x00	Disarm
0x01	Arm day/home zones only
0x02	Arm night/sleep zones only
0x03	Arm all zones
0x04 - 0xFF	Reserved

- sArmDisarmCode is an 8-character string containing the arm/disarm code (if a code is not required, set to "00000000")

- u8ZoneID is the identifier of the zone to arm/disarm

'Bypass' Command Payload The following structure contains the payload of a Bypass command.

```
typedef struct
{
    zuint8                  u8NumOfZones;
    zuint8                  *pu8ZoneID;
    tsZCL_CharacterString   sArmDisarmCode;
} tsCLD_IASACE_BypassPayload;
```

where:

- u8NumOfZones is the number of zones to be 'bypassed' (taken out of the system)
- pu8ZoneID is a pointer to a list of identifiers specifying the zones to be bypassed (the number of zones in the list is specified in u8NumOfZones)
- sArmDisarmCode is an 8-character string containing the arm/disarm code (if a code is not required, set to "00000000")

'Get Zone Information' Command Payload The following structure contains the payload of a Get Zone Information command.

```
typedef struct
{
    zuint8          u8ZoneID;
} tsCLD_IASACE_GetZoneInfoPayload;
```

where u8ZoneID is the identifier of the zone on which information is required.

'Set Bypassed Zone List' Command Payload The following structure contains the payload of a Set Bypassed Zone List command.

```
typedef struct
{
    zuint8          u8NumofZones;
    zuint8          *pu8ZoneID;
} tsCLD_IASACE_SetBypassedZoneListPayload;
```

where:

- u8NumofZones is the number of zones in the new bypassed zone list
- pu8ZoneID is a pointer to the new bypassed zone list (the number of zones in the list is specified in u8NumOfZones)

'Get Zone Status' Command Payload The following structure contains the payload of a Get Zone Status command.

```
typedef struct
{
    zuint8          u8StartingZoneID;
    zuint8          u8MaxNumOfZoneID;
    zbool           bZoneStatusMaskFlag;
```

(continues on next page)

(continued from previous page)

```
    zbmap16         u16ZoneStatusMask;
} tsCLD_IASACE_GetZoneStatusPayload;
```

where:

- u8StartingZoneID is the identifier of the first zone for which status information is required
- u8MaxNumOfZoneID is the maximum number of zones for which status information should be returned
- bZoneStatusMaskFlag is a Boolean indicating whether status information should be returned for all zones or only for those zones with particular status values (specified through u16ZoneStatusMask below):
  - TRUE - only zones with specific status values
  - FALSE - all zones
- u16ZoneStatusMask is a 16-bit bitmap indicating the zone status values of interest (used when bZoneStatusMaskFlag is set to TRUE) - the response to the request will contain information only for those zones with a status value indicated in this bitmap:

Bit	Description
0	Alarm1:

1 - Opened or alarmed

0 - Closed or not alarned

| |1|Alarm2:

1 - Opened or alarmed

0 - Closed or not alarned

| |2|Tamper:

1 - Tampered with

0 - Not tampered with

| |3|Battery:

1 - Low

0 - OK

| |4|Supervision reports:

1 - Reports

0 - No reports

| |5|Restore reports:

1 - Reports

0 - No reports

| |6|Trouble:

1 - Trouble/failure

0 - OK

| |7|AC (mains):

1 - Fault

0 - OK

| |8|Test mode:

1 - Sensor in test mode

0 - Sensor in operational mode

| |9|Battery defect:

1 - Defective battery detected

0 - Battery OK

| |10-15|Reserved|

'Panel Status Changed or Get Panel Status Response' The following structure contains the payload of a Panel Status Changed command or Get Panel Status Response.

```
typedef struct
{
    zenum8        ePanelStatus;
    zuint8       u8SecondsRemaining;
    zenum8        eAudibleNotification;
    zenum8        eAlarmStatus;
} tsCLD_IASACE_PanelStatusChangedOrGetPanelStatusRespPayload;
```

where:

- ePanelStatus is a value indicating the status to be displayed on the panel, as follows (enumerations are available in teCLD_IASACE_PanelStatus - see Section 38.8.2):

Value	Status
0x00	Disarmed (all zones) and ready to be armed
0x01	Armed stay
0x02	Armed night
0x03	Armed away
0x04	Exit delay
0x05	Entry delay
0x06	Not ready to be armed
0x07	In alarm
0x08	Arming stay
0x09	Arming night
0x0A	Arming away
0x0B - 0xFF	Reserved

- u8SecondsRemaining represents the time, in seconds, that the server will remain in the displayed state when the latter is 'Exit delay' or 'Entry delay' (for other states, this field should be set to 0x00).
- eAudibleNotification is a value specifying whether an audible notification (e.g. a chime) is required to signal a zone status change (enumerations are available in teCLD_IASACE_AudibleNotification - see Section 38.8.4):

Value	Status
0x00	Audible notification muted
0x01	Audible notification sounded
0x02 - 0xFF	Reserved

- eAlarmStatus is a value indicating the alarm status/type when the panel's state is 'In Alarm', as follows (enumerations are available in teCLD_IASACE_AlarmStatus - see Section 38.8.3):

Value	Status
0x00	No alarm
0x01	Burglar
0x02	Fire
0x03	Emergency
0x04	Police panic
0x05	Fire panic
0x06	Emergency panic
0x07 - 0xFF	Reserved

Parent topic: Structures

Event Data Structures The following structures hold the data contained in certain IAS ACE cluster events.

E_CLD_IASACE_CMD_ARM_RESP Data

```
typedef struct
{
    zenum8          eArmNotification;
} tsCLD_IASACE_ArmRespPayload;
```

where eArmNotification is an enumeration indicating the outcome of the Arm command, one of:

- E_CLD_IASACE_ARM_NOTIF_ALL_ZONES_DISARMED
- E_CLD_IASACE_ARM_NOTIF_ONLY_DAY_HOME_ZONES_ARMED
- E_CLD_IASACE_ARM_NOTIF_ONLY_NIGHT_SLEEP_ZONES_ARMED
- E_CLD_IASACE_ARM_NOTIF_ALL_ZONES_ARMED
- E_CLD_IASACE_ARM_NOTIF_INVALID_ARM_DISARM_CODE
- E_CLD_IASACE_ARM_NOTIF_NOT_READY_TO_ARM
- E_CLD_IASACE_ARM_NOTIF_ALREADY_DISARMED

E_CLD_IASACE_CMD_GET_ZONE_ID_MAP_RESP Data

```
typedef struct
{
    zbmap16  au16ZoneIDMap[CLD_IASACE_MAX_BYTES_FOR_NUM_OF_ZONES];
} tsCLD_IASACE_GetZoneIDMapRespPayload;
```

where au16ZoneIDMap[``] is an array, each element being a 16-bit bitmap indicating whether each of a set of zone identifiers is allocated - a Zone ID is represented by a single bit which is set to '1' if the identifier value has been allocated and '0' otherwise.

Array Element	Bit	Zone ID
au16ZoneIDMap[0]	01: 15	0x000x01: 0x0F
au16ZoneIDMap[1]	01: 15	0x100x11: 0x1F
:	:	:
au16ZoneIDMap[N]	01:n:15	16N16N + 0x1:16N + 0xn:16N + 0xF

E_CLD_IASACE_CMD_GET_ZONE_INFO_RESP Data

```
typedef struct
{
    zuint8                  u8ZoneID;
    zbmap16                 u16ZoneType;
    zieeeaddress            u64IeeeAddress;
    tsZCL_CharacterString   sZoneLabel;
} tsCLD_IASACE_GetZoneInfoRespPayload;
```

where:

- u8ZoneID is the identifier of the zone
- u16ZoneType is a value indicating the type of zone (for the possible values, refer to the description of the attribute e``16ZoneType of the IAS Zone cluster in Section 37.2)
- u64IeeeAddress is the IEEE/MAC address of the device which hosts the zone
- sZoneLabel is a character string representing a name/label for the zone

E_CLD_IASACE_CMD_ZONE_STATUS_CHANGED Data

```
typedef struct
{
    zuint8                  u8ZoneID;
    zenum16                 eZoneStatus;
    zenum8                  eAudibleNotification;
    tsZCL_CharacterString   sZoneLabel;
} tsCLD_IASACE_ZoneStatusChangedPayload;
```

where:

- u8ZoneID is the identifier of the zone
- u16ZoneType is a value indicating the type of zone (for the possible values, refer to the description of the attribute e``16ZoneType of the IAS Zone cluster in Section 37.2)
- eAudibleNotification is a value specifying whether an audible notification

  (e.g. a chime) to signal the change is required (enumerations are available in teCLD_IASACE_AudibleNotification - see Section 38.8.4):

Value	Status
0x00	Audible notification to be muted
0x01	Audible notification to be sounded
0x02 - 0xFF	Reserved

- sZoneLabel is a character string representing a name/label for the zone

E_CLD_IASACE_CMD_PANEL_STATUS_CHANGED Data

```
tsCLD_IASACE_PanelStatusChangedOrGetPanelStatusRespPayload
```

For details of this structure, see Section 38.7.5.

E_CLD_IASACE_CMD_GET_PANEL_STATUS_RESP Data

```
tsCLD_IASACE_PanelStatusChangedOrGetPanelStatusRespPayload
```

For details of this structure, see Section 38.7.5.

E_CLD_IASACE_CMD_BYPASS_RESP Data

```
typedef struct
{
    zuint8          u8NumofZones;
    zuint8         *pu8BypassResult;
} tsCLD_IASACE_BypassRespPayload;
```

where:

- u8NumOfZones is the number of zones 'bypassed' (taken out of the system)
- pu8BypassResult is a pointer to a list of identifiers specifying the zones bypassed (the number of zones in the list is specified in u8NumOfZones)

E_CLD_IASACE_CMD_GET_ZONE_STATUS_RESP Data

```
typedef struct
{
    zbool           bZoneStatusComplete;
    zuint8          u8NumofZones;
    zuint8         *pu8ZoneStatus;
} tsCLD_IASACE_GetZoneStatusRespPayload;
```

where:

- bZoneStatusComplete is a Boolean indicating whether the current response completes the set of zones for which status information can be returned (if not, the client should send another Get Zone Status command to the server):
  - TRUE - no more zone status information to be returned
  - FALSE - status information for more zones available to be queried
- u8NumofZones is the number of zones for which status information was returned in this response
- pu8ZoneStatus is a pointer to a list of status values for the reported zones (the number of values in the list is indicated by u8NumofZones above) - each is a 24-bit value containing the following information:

Bits	Description
0-7	Zone ID
8-23	Value of b16ZoneStatus attribute of the IAS Zone cluster for the zone

Parent topic: Structures

Parent topic:*IAS Ancillary Control Equipment Cluster*

Enumerations

teCLD_IASACE_ArmMode The following structure contains the enumerations used to indicate a mode of armament:

```
typedef enum
{
    E_CLD_IASACE_ARM_MODE_DISARM = 0x00,
    E_CLD_IASACE_ARM_MODE_ARM_DAY_HOME_ZONES_ONLY,
    E_CLD_IASACE_ARM_MODE_ARM_NIGHT_SLEEP_ZONES_ONLY,
    E_CLD_IASACE_ARM_MODE_ARM_ALL_ZONES,
} teCLD_IASACE_ArmMode;
```

Parent topic: Enumerations

teCLD_IASACE_PanelStatus The following structure contains the enumerations used to indicate the status of the panel:

```
typedef enum
{
    E_CLD_IASACE_PANEL_STATUS_PANEL_DISARMED = 0x00,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ARMED_DAY,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ARMED_NIGHT,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ARMED_AWAY,
    E_CLD_IASACE_PANEL_STATUS_PANEL_EXIT_DELAY,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ENTRY_DELAY,
    E_CLD_IASACE_PANEL_STATUS_PANEL_NOT_READY_TO_ARM,
    E_CLD_IASACE_PANEL_STATUS_PANEL_IN_ALARM,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ARMING_STAY,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ARMING_NIGHT,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ARMING_AWAY
} teCLD_IASACE_PanelStatus;
```

Parent topic: Enumerations

teCLD_IASACE_AlarmStatus The following structure contains the enumerations used to indicate the status/meaning of the alarm:

```
typedef enum
{
    E_CLD_IASACE_ALARM_STATUS_NO_ALARM = 0x00,
    E_CLD_IASACE_ALARM_STATUS_BURGLAR,
    E_CLD_IASACE_ALARM_STATUS_FIRE,
    E_CLD_IASACE_ALARM_STATUS_EMERGENCY,
    E_CLD_IASACE_ALARM_STATUS_POLICE_PANIC,
    E_CLD_IASACE_ALARM_STATUS_FIRE_PANIC,
    E_CLD_IASACE_ALARM_STATUS_EMERGENCY_PANIC
} teCLD_IASACE_AlarmStatus;
```

Parent topic: Enumerations

teCLD_IASACE_AudibleNotification The following structure contains the enumerations used to indicate the configuration of the audible indication:

```
typedef enum
{
    E_CLD_IASACE_AUDIBLE_NOTIF_MUTE = 0x00,
    E_CLD_IASACE_AUDIBLE_NOTIF_DEFAULT_SOUND
} teCLD_IASACE_AudibleNotification;
```

Parent topic: Enumerations

Parent topic:*IAS Ancillary Control Equipment Cluster*

Compile-time options To enable the IAS ACE cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_IASACE
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one of the following to the same file:

```
#define IASACE_SERVER
#define IASACE_CLIENT
```

The IAS ACE cluster contains macros that may be specified at compile-time by adding one or more of the following lines to the **zcl_options.h** file.

Maximum Size of Zone Table The maximum number of entries in a Zone table on the cluster server can be defined using the following line:

```
#define CLD_IASACE_ZONE_TABLE_SIZE n
```

where n is the desired maximum (e.g. 8).

Maximum Length of Arm/Disarm Code The maximum length of string allowed for the arm/disarm code can be defined using the following line:

```
#define CLD_IASACE_MAX_LENGTH_ARM_DISARM_CODE n
```

where n is the desired maximum.

Maximum Length of Zone Label The maximum length of string allowed for a zone name/label can be defined using the following line:

```
#define CLD_IASACE_MAX_LENGTH_ZONE_LABEL n
```

where n is the desired maximum.

Disable APS Acknowledgements for Bound Transmissions APS acknowledgements for bound transmissions from this cluster can be disabled using the following line:

```
#define CLD_IASACE_BOUND_TX_WITH_APS_ACK_DISABLED
```

Cluster Revision Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_IASACE_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*IAS Ancillary Control Equipment Cluster*

IAS Warning Device Cluster

This chapter describes the IAS Warning Device (WD) cluster which provides an interface to a Warning Device in an IAS (Intruder Alarm System).

The IAS WD cluster has a Cluster ID of 0x0502.

Overview

The IAS WD cluster provides an interface to an IAS Warning Device, allowing warning indications triggered by alarm conditions to be sent to it. The server side of the cluster is implemented on the IAS Warning Device and the client side is implemented on the triggering device. The IAS Warning Device is detailed in the *ZigBee Devices User Guide (JNUG3131)*.

To use the functionality of this cluster, you must include the file **IASWD.h** in your application and enable the cluster by defining CLD_IASWD in the **zcl_options.h** file.

The inclusion of the client or server software must be pre-defined in the compile-time options of the application. In addition, if the cluster resides on a custom endpoint, then the role of client or server must also be specified when creating the cluster instance.

The compile-time options for the IAS WD cluster are fully detailed in Section 39.7.

Parent topic:*IAS Warning Device Cluster*

IAS WD Structure and Attribute

The structure definition for the IAS WD cluster is:

```
typedef struct
{
#ifdef IASWD_SERVER
  zuint16   u16MaxDuration;
#endif
  zuint16   u16ClusterRevision;
} tsCLD_IASWD;
```

where:

- u``16MaxDuration is the maximum duration, in seconds, for which the alarm can be continuously active (for example, a siren sounded). The range of possible values is 0 to 65534 seconds and the default value is 240 seconds.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*IAS Warning Device Cluster*

Issuing Warnings

The IAS WD cluster allows a device which detects warning conditions (example, fire) to trigger a warning on an IAS Warning Device which, in turn, initiates a physical alarm such as a siren and/or strobe. The IAS Warning Device hosts the cluster server and the triggering device hosts the cluster client.

Two types of warning can be initiated:

- **Warning mode:** This mode indicates a genuine emergency, such as a fire or an intruder. On detection of the emergency condition, the application on the triggering device must call the **eCLD_IASWDStartWarningReqSend()** function, which sends a Start Warning command to the Warning Device. The payload of this command contains the time-duration for which

the Warning Device must remain in warning mode. The specified duration must not exceed the maximum duration defined in the u16MaxDuration attribute on the Warning Device (see Section 39.2). The payload also contains details of the warning and the strobe requirements, if any. On receiving this command, an E_CLD_IASWD_CMD_WD_START_WARNING event is generated on the Warning Device (see Section 39.4) for the attention of the application.

- **Squawk mode:** This mode indicates a change of state of the IAS system - that is, armed or disarmed. Thus, this is typically a short audible beep or 'squawk' that is emitted when the system is armed or disarmed. To initiate a squawk, the application on the triggering device must call the function **eCLD_IASWDSquawkReqSend()**, which sends a Squawk command to the Warning Device. The payload also contains details of the squawk and the strobe requirements, if any. On receiving this command, an E_CLD_IASWD_CMD_WD_SQUAWK event is generated on the Warning Device (see Section 39.4) for the attention of the application.

The payloads of the commands are detailed in Section 39.6.2.

Note: In order to maintain timing information on the cluster server, the application on the Warning Device must periodically call the **eCLD_IASWDUpdate()** function every 100 ms. These calls can be prompted using a software timer.

Note: The u16MaxDuration attribute on the Warning Device can be updated by the application on this device by calling the function **eCLD_IASWDUpdateMaxDuration()**.

Parent topic:*IAS Warning Device Cluster*

IAS WD Events The IAS WD cluster has its own events that are handled through the callback mechanism outlined in *Chapter 3*. If a device uses the IAS WD cluster then IAS WD event handling must be included in the callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function (for example, through **eHA_RegisterWarningDeviceEndPoint()** for a Warning Device). The relevant callback function is then invoked when an IAS WD event occurs.

For an IAS WD event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_IASWDCallBackMessage structure:

```
typedef struct
{
  uint8   u8CommandId;
  union
  {
  tsCLD_IASWD_StartWarningReqPayload  *psWDStartWarningReqPayload;
  tsCLD_IASWD_SquawkReqPayload   *psWDSquawkReqPayload;
  tsCLD_IASWD_StrobeUpdate  *psStrobeUpdate;  /* Internal */
  tsCLD_IASWD_WarningUpdate   *psWarningUpdate; /* Internal */
  } uMessage;
} tsCLD_IASWDCallBackMessage;
```

When an IAS WD event occurs, one of several command types could have been received. The relevant command type is specified through the u8CommandId field of the tsSM_CallBackMessage structure. The possible command/event types are detailed in the table below (not that psStrobeUpdate and psWarningUpdate are for internal use only).

u8Comman Enumera-tion	Description
E_CLD_IASV	A Start Warning command has been received by the cluster server - this command requests that the alarm is activated for a specified time. The command payload is contained in the event in the tsCLD_IASWD_StartWarningReqPayload structure, described in Section 39.6.2.
E_CLD_IASV	A Squawk command has been received by the cluster server - this command requests that the alarm is briefly activated to emit a 'squawk' to indicate a status change, such as system disarmed. The command payload is contained in the event in the tsCLD_IASWD_SquawkReqPayload structure, described in Section 39.6.2.

Parent topic:*IAS Warning Device Cluster*

Functions The following IAS WD cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_IASWDCreateIASWD
2. eCLD_IASWDUpdate
3. eCLD_IASWDUpdateMaxDuration
4. eCLD_IASWDStartWarningReqSend
5. eCLD_IASWDSquawkReqSend

eCLD_IASWDCreateIASWD

```
teZCL_Status eCLD_IASWDCreateIASWD(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits,
        tsCLD_IASWDCustomDataStructure
        *psCustomDataStructure);
```

Description This function creates an instance of the IAS WD cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates an IAS WD cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the IAS WD cluster. This parameter can refer to a pre-filled structure called sCLD_``IASWD which is provided in the **IASWarningDevice.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_IASWD which defines the attributes of IAS WD cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster.
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 39.6.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_IASWDUpdate

```
teZCL_Status eCLD_IASWDUpdate(
uint8 u8SourceEndPoint);
```

Description This function can be used on an IAS WD cluster server to update the timing requirements of the Warning Device. The function should be called by the application at a rate of once every 100 ms.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS WD cluster resides

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Functions

eCLD_IASWDUpdateMaxDuration

```
teZCL_Status eCLD_IASWDUpdateMaxDuration(
uint8 u8SourceEndPointId,
uint16 u16MaxDuration);
```

Description This function can be used on an IAS WD cluster server to set the value of the u16MaxDuration attribute, which represents the maximum duration, in seconds, for which the alarm can be continuously active.

The set value is the maximum duration, in seconds, for which the alarm can be active following a received Start Warning request.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS WD cluster resides
- *u16MaxDuration*: Value to which attribute will be set, in the range 0 to 65534

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Functions

eCLD_IASWDStartWarningReqSend

```
teZCL_Status eCLD_IASWDStartWarningReqSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_IASWD_StartWarningReqPayload *psPayload);
```

Description This function can be used on IAS WD cluster client to send a Start Warning command to the IAS WD server on a Warning Device.

The receiving IAS WD server activates the alarm on the Warning Device for a specified duration.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 39.6.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

eCLD_IASWDSquawkReqSend

```
teZCL_Status eCLD_IASWDSquawkReqSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_IASWD_SquawkReqPayload *psPayload);
```

Description This function can be used on IAS WD cluster client to send a Squawk command to the IAS WD server on a Warning Device.

The receiving IAS WD server will briefly activate the alarm on the Warning Device to emit a 'squawk' - depending on the device, this could be a visible and/or audible emission. The parameters of the squawk are specified in the command payload.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 39.6.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

Parent topic: Functions

Parent topic:*IAS Warning Device Cluster*

Structures

Custom Data Structure The IAS WD cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
 tsCLD_IASWD_SquawkReqPayload sSquawk;
 tsCLD_IASWD_StartWarningReqPayload sWarning;
 uint32   u32WarningDurationRemainingIn100MS;
 tsZCL_ReceiveEventAddress  sReceiveEventAddress;
 tsZCL_CallBackEvent  sCustomCallBackEvent;
 tsCLD_IASWDCallBackMessage sCallBackMessage;
} tsCLD_IASWD_CustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Custom Command Payloads The following structures contain the payloads for the IAS WD cluster custom commands.

'Start Warning' Payload The following structure contains the payload of a Start Warning command.

```
typedef struct
{
 uint8   u8WarningModeStrobeAndSirenLevel;
 uint16  u16WarningDuration;
 uint8   uStrobeDutyCycle;
 enum8       eStrobeLevel;
}tsCLD_IASWD_StartWarningReqPayload;
```

where:

- u8WarningModeStrobeAndSirenLevel is an 8-bit bitmap containing the requirements for the warning alarm, as follows:

Bits	Description
0-3	Warning Mode - indicates the meaning of the requested warning:0 - Stop (no warning)1 - Burglar2 - Fire3 - Emergency4 - Police panic5 - Fire panic6 - Emergency (medical) panic All other values are reserved
4-5	Strobe* - indicates whether a visual strobe indication of the warning is required:0 - No strobe1 - Use strobeOther values are reserved
6-7	Siren Level - indicates the requested level of an audible siren (if enabled):0 - Low level1 - Medium level2 - High level3 - Very high level

* If 'Strobe' is 1 and 'Warning Mode' is 0, only the strobe is activated

- u16WarningDuration is the requested time-duration of the warning, in seconds, which must be less than or equal to the value of the u16MaxDuration attribute
- uStrobeDutyCycle is the duty-cycle of the strobe pulse, expressed as a percentage in 10% steps (example, 0x1E represents 30%) - invalid values are rounded to the nearest multiple of 10%
- eStrobeLevel is the level of the strobe (pulse)

'Squawk' Payload The following structure contains the payload of a Squawk command.

```
typedef struct
{
  uint8  u8SquawkModeStrobeAndLevel;
}tsCLD_IASWD_SquawkReqPayload;
```

Where u8SquawkModeStrobeAndLevel is an 8-bit bitmap containing the requirements for the 'squawk', as follows.

Bits	Description
0-3	Squawk Mode - indicates the meaning of the required 'squawk':0 - System is armed1 - System is disarmedAll other values are reserved
4	Strobe - indicates whether a visual strobe indication of the 'squawk' is required:0 - No strobe1 - Use strobe
5	Reserved
6-7	Squawk Level - indicates the requested level of the audible squawk sound:0 - Low level1 - Medium level2 - High level3 - Very high level

Parent topic: Structures

Event Data Structures The following structures hold the data contained in certain IAS WD cluster events.

E_CLD_IASWD_CLUSTER_UPDATE_STROBE Data

```
typedef struct
{
 bool_t    bStrobe;
 uint8     u8StrobeDutyCycle;
 zenum8    eStrobeLevel;
}tsCLD_IASWD_StrobeUpdate;
```

where:

```
bStrobe is the current (new) status of the strobe:
```

- TRUE - Strobe 'on'
- FALSE - Strobe 'off'
  - uStrobeDutyCycle is the duty-cycle of the strobe pulse, expressed as a percentage in 10% steps (example, 0x1E represents 30%) - invalid values is rounded to the nearest multiple of 10%
  - eStrobeLevel is the level (brightness) of the strobe pulse:
- 0 - Low level
- 1 - Medium level
- 2 - High level
- 3 - Very high level
  - All other values are reserved

E_CLD_IASWD_CLUSTER_UPDATE_WARNING Data

```
typedef struct
{
 uint8     u8WarningMode;
 uint16    u16WarningDurationRemaining;
 zenum8    eStrobeLevel;
}tsCLD_IASWD_WarningUpdate;
```

where:

- u8WarningMode is a value indicating the current warning mode:
  - 0 - No warning
  - 1 - Burglar
  - 2 - Fire
  - 3 - Emergency
  - 4 - Police panic
  - 5 - Fire panic
  - 6 - Emergency (medical) panic
    - All other values are reserved

- u16WarningDurationRemaining is the time, in seconds, during which the device remains in warning mode
- eStrobeLevel is the level of the strobe (pulse)

Parent topic: Structures

Parent topic:*IAS Warning Device Cluster*

Compile-time Options To enable the IAS WD cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_IASWD
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one of the following to the same file:

```
#define IASWD_SERVER
#define IASWD_CLIENT
```

The IAS WD cluster contains macros that may be specified at compile-time by adding one or more of the following lines to the **zcl_options.h** file.

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_IASWD_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*IAS Warning Device Cluster*

Part IX: Smart Energy Clusters This part comprises three chapters:

- *Chapter 40* details the **Price**cluster
- *Chapter 41* details the **Demand-Response and Load Control**cluster
- *Chapter 42* details the **Simple Metering** cluster

Price Cluster This chapter outlines the Price cluster, which is used to hold and exchange price information.

The Price cluster has a Cluster ID of 0x0700.

CAUTION:

Important: While the Price cluster software supports Block mode, this mode is not certifiable in SE 1.1.1 (07-5356-17) or earlier and is therefore not fully documented in this chapter. Customers who wish to use Block mode should contact NXP for direct support.

Overview The Price cluster is required in ZigBee devices as indicated in the table below.

	Server-side	Client-side
Mandatory in...	ESP	Smart Appliance
Optional in...		ESP Metering Device, IPD, PCT, Load Control Device

The ESP normally acts as the Price cluster server, holding price information received from the utility company. Other devices act as clients and receive price information from the ESP. The clients' price information must be kept up-to-date with the server's price information.

The Price cluster is enabled by defining CLD_PRICE in the **zcl_options.h** file. Further compile-time options for the Price cluster are detailed in Section 40.13.

The Price cluster can operate in a mode in which pricing is based on the time at which the consumption occurs - this is called Time-Of-Use (TOU) mode. The cluster allows up to fifteen price 'tiers', numbered 1 to 15, which correspond to different time periods. Each price tier is given a label, which is used to identify the tier - typical labels are "Normal", "Shoulder", "Peak", "Real-time Pricing" and "Critical Peak". The tiers must be numbered consecutively in price order, with Tier 1 being the cheapest.*Note that tiers 7 to 15 are not certifiable in SE 1.1.1 or earlier and are reserved for future use.*

The information that can potentially be stored in the Price cluster is organised into the following attribute sets: Tier Label, Block Threshold, Block Period, Commodity, Block Price Information, Billing Period Information. The attribute sets Block Threshold, Block Period, Block Price Information and Billing Period Information are reserved for future use (with Block mode). There is also a set of attributes exclusively for use on a Price cluster client.

The cluster includes commands for requesting and publishing (distributing) price information. The price information that is valid for a certain time is sent from the Price cluster server (ESP) to the Price cluster clients using **Publish Price**commands, which may be sent from the ESP under the following circumstances:

- Unsolicited from the server - for example, when new pricing information has been received from the utility company or a new price tier becomes active
- In response to a **Get Current Price** command, sent by a client that needs the price for the current time period
- In response to a **Get Scheduled Prices** command, sent by a client that needs both current and future prices

Functions are provided for implementing the cluster commands. These functions are referenced in Section 40.4 and Section 40.5, and detailed in Section 40.9.

Parent topic:*Price Cluster*

Price cluster structure and attributes The Price cluster is contained in the following tsCLD_Price structure:

```
typedef struct CLD_Price_tag
{
    /* Tier Price Label Set (D.4.2.2.1) */
#if (CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT != 0)
 tsZCL_OctetString  asTierPriceLabel[CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT];
 uint8  au8TierPriceLabel[CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT][SE_PRICE_SERVER_
↪MAX_STRING_LENGTH];
#endif
 /* Block Threshold Set (D.4.2.2) */
#if (CLD_P_ATTR_BLOCK_THRESHOLD_MAX_COUNT != 0)
 zuint48        au48BlockThreshold[CLD_P_ATTR_BLOCK_THRESHOLD_MAX_COUNT];
#endif
    /* Block Period Set (D.4.2.2.3) */
#ifdef CLD_P_ATTR_START_OF_BLOCK_PERIOD
    zutctime            utctStartOfBlockPeriod;
#endif
#ifdef CLD_P_ATTR_BLOCK_PERIOD_DURATION
    zuint24             u24BlockPeriodDuration;
#endif
#ifdef CLD_P_ATTR_THRESHOLD_MULTIPLIER
```

(continues on next page)

(continued from previous page)

```
    zuint24            u24ThresholdMultiplier;
#endif
#ifdef CLD_P_ATTR_THRESHOLD_DIVISOR
    zuint24            u24ThresholdDivisor;
#endif
    /* Commodity Set Set (D.4.2.2.4) */
#ifdef CLD_P_ATTR_COMMODITY_TYPE
    zenum8             e8CommodityType;
#endif
#ifdef CLD_P_ATTR_STANDING_CHARGE
    zuint32            u32StandingCharge;
#endif
#ifdef CLD_P_ATTR_CONVERSION_FACTOR
    zuint32            u32ConversionFactor;
#endif
#ifdef CLD_P_ATTR_CONVERSION_FACTOR_TRAILING_DIGIT
    zbmap8             b8ConversionFactorTrailingDigit;
#endif
#ifdef CLD_P_ATTR_CALORIFIC_VALUE
    zuint32            u32CalorificValue;
#endif
#ifdef CLD_P_ATTR_CALORIFIC_VALUE_UNIT
    zenum8             e8CalorificValueUnit;
#endif
#ifdef CLD_P_ATTR_CALORIFIC_VALUE_TRAILING_DIGIT
    zbmap8             b8CalorificValueTrailingDigit;
#endif
    /* Block Price Information Set (D.4.2.2.5) */
#if (CLD_P_ATTR_NO_TIER_BLOCK_PRICES_MAX_COUNT != 0)
    zuint32     au32NoTierBlockPrice[CLD_P_ATTR_NO_TIER_BLOCK_PRICES_MAX_COUNT];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 0)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
    zuint32     au32Tier1BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 1)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
    zuint32     au32Tier2BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 2)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
    zuint32    au32Tier3BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 3)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
  zuint32  au32Tier4BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 4)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
  zuint32     au32Tier5BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 5)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
  zuint32    au32Tier6BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 6)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
  zuint32   au32Tier7BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 7)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
```

(continues on next page)

(continued from previous page)

```
  zuint32   au32Tier8BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 8)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
 zuint32     au32Tier9BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 9)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
   zuint32     au32Tier10BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 10)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
  zuint32     au32Tier11BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 11)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
  zuint32         au32Tier12BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 12)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
 zuint32        au32Tier13BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 13)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
  zuint32         au32Tier14BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 14)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_
↪EACH_TIER_PRICE != 0))
  zuint32     au32Tier15BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#ifdef CLD_P_ATTR_START_OF_BILLING_PERIOD
   zutctime          utctStartOfBillingPeriod;
#endif
#ifdef CLD_P_ATTR_BILLING_PERIOD_DURATION
   zuint24           u24BillingPeriodDuration;
#endif
#ifdef CLD_P_CLIENT_ATTR_PRICE_INCREASE_RANDOMIZE_MINUTES
   uint8             u8ClientIncreaseRandomize;
#endif
#ifdef CLD_P_CLIENT_ATTR_PRICE_DECREASE_RANDOMIZE_MINUTES
   uint8             u8ClientDecreaseRandomize;
#endif
#ifdef CLD_P_CLIENT_ATTR_COMMODITY_TYPE
   zenum8            e8ClientCommodityType;
#endif
} tsCLD_Price;
```

where:

'Tier Label' Attribute Set

- The following are optional attributes that are only relevant to TOU mode**(*tiers 7 to 15 are not certifiable in SE 1.1.1 or earlier and are reserved for future use*):
  - asTierPriceLabel[CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT]

    is a tsZCL_OctetString structure containing information on tier labels. The maximum size of asTierPriceLabel is defined by assigning a value to CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT. This optional element is paired with au8TierPriceLabel (below)

- au8TierPriceLabel[CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT]``[SE_PRICE_SERVER_MA is an array containing the tier labels, e.g. "Peak". This optional element is paired with the element asTierPriceLabel (above)

Note: Memory space for each (enabled) price tier label is statically allocated and comprises 13 bytes per label (plus one byte for the 'octet count'). Therefore, memory space remains allocated for unused bytes.

'Block Threshold' Attribute Set

- The following are optional attributes that relate to Block mode and are fully described in the *ZigBee Smart Energy Profile Specification*(*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):

```
au48BlockThreshold[CLD_P_ATTR_BLOCK_THRESHOLD_MAX_COUNT]
```

'Block Period' Attribute Set

- The following are optional attributes that relate to Block mode and are fully described in the *ZigBee Smart Energy Profile Specification*(*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - utctStartOfBlockPeriod
  - u24BlockPeriodDuration
  - u24ThresholdMultiplier
  - u24ThresholdDivisor

'Commodity' Attribute Set

- The following are optional attributes:
  - e8CommodityType is an enumeration representing the type of commodity (e.g. gas) to which the prices apply - the enumerations used are those provided in the teCLD_SM_MeteringDeviceType structure of the Simple Metering cluster and listed in Section 42.10.6
  - u32StandingCharge is the value of a fixed daily 'standing charge' associated with supplying the commodity, expressed in the currency and with the decimal places indicated in the Publish Price command described in Section 40.11.1 (the value 0xFFFFFFFF indicates that the field is not used)
  - u32ConversionFactor is used only for gas and accounts for the variation of gas volume with temperature and pressure (and is dimensionless). The Price server can change this conversion factor at any time and this attribute contains the currently active value. The default value is 1. The position of the decimal point is indicated by b8``ConversionFactorTrailingDigit described below.
  - b8``ConversionFactorTrailingDigit is an 8-bit bitmap which indicates the location of the decimal point in the u32ConversionFactor attribute. The most significant 4 bits indicate the number of digits after the decimal point. The remaining bits are reserved.
  - u32CalorificValue is used only for gas and indicates the quantity of energy in MJ that is generated per unit volume or unit mass of gas burned (see e8``CalorificValueUnit) - the value can be used to calculate energy consumption in kWh. The position of the decimal point is indicated by b8``CalorificValueTrailingDigit described below.

- e8``CalorificValueUnit is an enumerated value indicating whether u32CalorificValue is quantified per unit volume or per unit mass. The possible values are 0x01 for MJ/m3 and 0x02 for MJ/kg (all other values are reserved).
- b8``CalorificValueTrailingDigit is an 8-bit bitmap which indicates the location of the decimal point in the u32CalorificValue attribute. The most significant 4 bits indicate the number of digits after the decimal point. The remaining bits are reserved.

'Block Price Information' Attribute Set

- The following are optional attributes that relate to Block mode and are fully described in the *ZigBee Smart Energy Profile Specification*(*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):

```
au32NoTierBlockPrice[CLD_P_ATTR_NO_TIER_BLOCK_PRICES_MAX_COUNT]
au32Tier1BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier2BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier3BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier4BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier5BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier6BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier7BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier8BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier9BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier10BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier11BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier12BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier13BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier14BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
au32Tier15BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]
```

'Billing Period Information' Attribute Set

- The following are optional attributes that relate to Block mode (*both attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):

```
utctStartOfBillingPeriod
u24BillingPeriodDuration
```

Client Attribute Set

- The following set of attributes are only for use on a Price cluster client:
  - u8ClientIncreaseRandomize represents the maximum length of time, in minutes, between a client node applying a price increase and taking a resulting action (such as reducing its power consumption). The action may be performed before or after the price increase is implemented, and the delay (either way) must be chosen at random by the application on the node. The maximum is set in minutes, in the range 0 to 60 minutes, but it is recommended that the random delay is selected in seconds.
  - u8ClientDecreaseRandomize represents the maximum length of time, in minutes, between a client node applying a price decrease and taking a resulting action (such as switching itself on). The action may be performed before or after the price decrease is implemented, and the delay (either way) must be chosen at random by the application on the node. The maximum is set in minutes, in the range 0 to 60 minutes, but it is recommended that the random delay is selected in seconds.

– e8ClientCommodityType is an enumeration representing the commodity that is priced on the client device. This enumeration is one from the 'Metering Device Type' enumerations listed in Table74.

Note: Price information for Time-Of-Use (TOU) mode is held in the tsSE_PricePublishPriceCmdPayload structure described in Section 40.11.1. Prices are matched to tiers using the strings defined in the Tier Label attributes.

Parent topic:*Price Cluster*

Attribute settings The Price cluster structure (see Section 40.2) contains no mandatory elements. All elements are optional, each being enabled/disabled through a corresponding macro defined in the **zcl_options.h** file - for example, the commodity type attribute is enabled/disabled through the macro CLD_P_ATTR_COMMODITY_TYPE. The attributes that are used depend on the number of tiers implemented (and Block mode attributes must be disabled).

Note:

- The Tier Label attributes are connected to the tier-related attributes in the Simple Metering cluster, e.g. u48CurrentTier6SummationDelivered for Tier 6. For a complete list of these Simple Metering attributes, refer to Section 42.2.
- The price information for Time-Of-Use (TOU) mode is stored in the structure tsSE_PricePublishPriceCmdPayload described in Section 40.11.1.

Parent topic:*Price Cluster*

Initializing and maintaining price lists A list of prices is held on both the Price cluster server (ESP) and client(s). The price list on a client must be maintained to mirror the price list on the server. On device startup, the Price cluster software initializes the device's price list as empty. The price lists are then built and maintained as described below.

The ESP receives price information from the utility company and populates its price list with this information. The application on the ESP does this by calling the function **eSE_PriceAddPriceEntry()** for each new price received from the utility company. This function also sends out a Publish Price command containing the new price information to all Price cluster clients in the network. On receiving this command, a Price cluster client automatically adds this price information to its own price list (see Section 40.5.1). However, at ESP startup, there may be no other active nodes in the network to receive the Publish Price commands (since the ESP is normally also the ZigBee Co-ordinator and is, therefore, the first node to be started). For this reason, the Price cluster clients should normally request the scheduled prices from the ESP when they start up, as described in Section 40.5.3.

Note: When initializing the price list at ESP startup, the ESP application should call **eSE_PriceAddPriceEntry()** with the address mode parameter set to E_ZCL_AM_NO_TRANSMIT, so that the price additions are not subsequently transmitted.

Note: A Price cluster server should take precautions to prevent clients from attempting to read the server price list during ESP initialization, before the prices have been received from the utility company. This can be achieved by adding the obtained prices to the server price list after the call to the relevant endpoint registration function (for example, **eSE_RegisterEspEndPoint()**) but before the call **to ZPS_eAplZdoStartStack()**.

A price list is maintained in time order and if there is an active price, it is positioned at the head of the list (with index 0). Price lists on clients are updated to reflect the price list on the server, as described in Section 40.5.

Note: The Price cluster of ZigBee Smart Energy automatically deletes a price entry from a client or server price list immediately after the price event has expired. This is because the start-time of a price event is a universal time (UTC) and therefore corresponds to a one-off event. In practice, the price list may need a new price schedule daily, which may be provided by the utility company.

Alternatively, if a similar schedule is required every day, the ESP application can keep a local copy of the schedule, which it can modify (e.g. start-times) and add to the price list on a daily basis.

The active price is always at the head of the price list (entry zero). The application should check that the entry at the head of the list is active before displaying it as the current price. If it is not active, a message may be displayed indicating that the current price is not known. The item at the head of the list is active if both of the following hold:

- Its start time is less than or equal to the current time, obtained by **u32ZCL_GetUTCTime()**
- The time on the client has been synchronized, i.e. a call to **bZCL_GetTimeHasBeenSynchronised()** returns TRUE

In addition to the function **eSE_PriceAddPriceEntry()**, the following functions allow an ESP application to access and manipulate its price list:

- **eSE_PriceGetPriceEntry()** obtains the price entry with the specified index.
- **eSE_PriceDoesPriceEntryExist()** checks whether there is a price entry with the specified start-time.
- **eSE_PriceRemovePriceEntry()** deletes the price entry with the specified

  start-time.
- **eSE_PriceClearAllPriceEntries()** deletes all price entries in the list.

These functions are fully detailed in Section 40.9.

Parent topic:*Price Cluster*

Publishing price information This section and its sub-sections describe the ways in which price information can be published (distributed) in a ZigBee network. As introduced in Section 40.1, there are three ways in which price information may be published to the network from the Price cluster server (ESP):

- Unsolicited unicasts - refer to Section 40.5.1
- Response to a Get Current Price command - refer to Section 40.5.2
- Response to a Get Scheduled Prices command - refer to Section 40.5.3

All of the above methods require the ESP to send a Publish Price command to the relevant device(s), where the payload of this command includes information such as resource (for example, gas), unit of measure, currency, price, current time, start-time, and duration. On receipt of this command, if valid, the received price information is automatically added to the price list on the device. If it is successfully added, an E_SE_PRICE_TABLE_ADD event is generated on the receiving device and this event is handled by the callback function registered for the relevant endpoint.

Unsolicited Price Updates When the ESP receives updated price information from the utility company (via the backhaul network) or a new price tier becomes active, the ESP must inform all network devices that are using the Price cluster. The ESP therefore individually unicasts a Publish Price command to all these devices. This command is sent out automatically - there is no need for the application on the ESP to explicitly send the command. In the case of new prices received from the utility company, the ESP application must call the function **eSE_PriceAddPriceEntry()** to add the new price to the price list held by the server, and the Publish Price command is then automatically sent out (possibly with a 'start-time of now'). Note that if the stack has not been started when **eSE_PriceAddPriceEntry()** is called, the function's address mode parameter should be set to E_ZCL_AM_NO_TRANSMIT, so that no transmission is attempted.

It is recommended that price updates on the ESP are relayed to Price cluster clients with which the ESP has been (previously) bound.

Note: Each of these bindings is initiated on the client node (e.g. IPD) using the ZigBee PRO stack function **ZPS_eAplZdpBindUnbindRequest()** to add the client's address and endpoint to the Binding table on the ESP. Binding is described in the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

Therefore, when updating its price list, the ESP application should call **eSE_PriceAddPriceEntry()** with the address mode parameter set to E_ZCL_AM_BOUND, so that the price updates are transmitted only to bound endpoints/nodes.

As an alternative to using binding, the ESP can maintain a list of network nodes that are able to receive unsolicited Publish Price commands at all times - that is, nodes with radio receivers that remain active during idle periods (e.g. when sleeping). Unsolicited updates are then only sent to clients in this group. The ESP gathers information for this group from the Get Current Price commands received from clients (see Section40.5.2). This option requires the address node parameter to be set to ZPS_E_APL_AF_BROADCAST_RX_ON in **eSE_PriceAddPriceEntry()**.

The ESP can send unsolicited Publish Price commands with 'start-time of now' when an E_SE_PRICE_TABLE_ACTIVE event indicates that a new price has become active (see Section 40.8). This command can be used by devices that do not implement a real-time clock.

Parent topic: Publishing price information

Get Current Price Any device which supports the Price cluster can request the currently active price information from the ESP by sending a Get Current Price command. The function **eSE_PriceGetCurrentPriceSend()** allows a Price cluster client to send this command to the Price cluster server and deal with the response.

- On receiving the command, the server automatically responds with a Publish Price command containing the requested price information.
- On receiving the response, the client checks whether the received price information is currently in the client's price list. If it is not, the client adds the new price information to the list and generates an E_SE_PRICE_TABLE_ADD event - this event is handled by the callback function registered for the relevant endpoint.

The Get Current Price command contains information on whether the radio receiver of the sending device remains active when the node is otherwise idle (e.g. sleeping). If this is true, the ESP application can use the address of the node to update a list of such devices, which it may use when sending out unsolicited Publish Price commands (see Section 40.5.1). The ESP application can extract this information from the event E_SE_PRICE_GET_CURRENT_PRICE_RECEIVED which is generated when a Get Current Price command is received by the server - this event is handled by the callback function registered for the relevant endpoint.

Parent topic: Publishing price information

Get Scheduled Prices Any device which supports the Price cluster can request the current price schedule from the ESP by sending a Get Scheduled Prices command - the schedule includes a set of prices with their start-times and durations. The function **eSE_PriceGetScheduledPricesSend()** allows a Price cluster client to send this command to the Price cluster server and deal with the responses.

- On receiving the command, the server automatically responds with a sequence of Publish Price commands, where each of these responses contains the information for one scheduled price.
- On receiving a response, the client checks whether the received price information is currently in the client's price list. If it is not, the client adds the new price information to the list and generates an E_SE_PRICE_TABLE_ADD event - this event is handled by the callback function registered for the relevant endpoint.

Parent topic: Publishing price information

Parent topic:*Price Cluster*

Time-synchronization via Publish Price commands As an alternative to using the Time cluster to time-synchronize a ZigBee device with the ESP (as described in Section 18.5.3), the local application can use the time embedded in a Publish Price command from the ESP (see Section 40.5), as described below.

Note: A device that implements the Price cluster must also implement the Time cluster.

It is the responsibility of the application on a ZigBee device to perform time-synchronization with the ESP. This involves updating the ZCL time on the local device.

The initialization of the ZCL time on a device should be performed using the Time cluster by requesting the current time from the ESP, as described in Section 18.5.2 (this method also gets the time-zone and daylight saving information).

Subsequent re-synchronizations of a device with the time-master can use the time contained in Publish Price commands from the ESP (but note that no time-zone or daylight saving information is included). Therefore, a device can update its ZCL time whenever it receives a Publish Price command. On receiving this command, a 'data indication' stack event is generated, which causes a ZCL user task to be activated. The event is initially handled by this task as described in *Chapter 3*, resulting in an E_ZCL_ZIGBEE_EVENT event being passed to the ZCL via **vZCL_EventHandler()**. The ZCL invokes the relevant user-defined callback function (see *Chapter 3*) which, provided that the event is of the type E_SE_PRICE_TIME_UPDATE, must update the ZCL time using **vZCL_SetUTCTime()**.

Note that the utctTime field of the local copy of the Time cluster is not updated, since this should only be done following a read of the Time cluster attributes from the server.

CAUTION:

If a device is handling Publish Price commands from more than one server, the time must only be updated with time events from one server, to prevent the time from jittering forwards and backwards if the servers' times are not in sync.

The time-synchronization of a device (with the time-master) should be performed regularly. As a rule, if no Publish Price commands have been received from the ESP in the last 48 hours, the device should request the current time from the ESP and update its own times as described in Section 18.5.3.

It is worth noting that an undefined ZCL time causes the following issues in the Price cluster:

- A Price cluster server without a ZCL time cannot issue any Publish Price commands, since the current time is a mandatory field of this command.
- A Price cluster client without a ZCL time cannot process a Publish Price command with a 'start-time of now', unless the ZCL time is first set with the time extracted from the received command.
- If the price at the head of the price list has a specified start-time, it is not possible to know whether this price is active or not.

Regarding the last point, a device should be time-synchronized with the ESP (as described in Section 18.5.2) before an attempt is made to add scheduled prices to the device's price list. Then, if the device receives a scheduled price with a 'start-time of now', it is permissible to add this price to the list.

Parent topic:*Price Cluster*

Conversion factor and calorific value (gas only) The Price cluster provides attributes related to conversion factor and calorific value for use with gas (only):

- **Conversion factor:** Accounts for the variation of gas volume with temperature and pressure
- **Calorific value:** Indicates the quantity of energy in MJ that is generated per unit volume or unit mass of gas burned

The attributes associated with the above properties are part of the 'Commodity' set - refer to Section 40.2.

If required, conversion factor and/or calorific value must be enabled in the compile-time options, as described in Section 40.13.

Conversion factors and calorific values can be independently scheduled with associated start-times. The Price cluster server (ESP) and clients each maintain a list of the scheduled conversion factors and a list of the scheduled calorific values (along with their start-times). The maximum number of entries in each list is by default 2 (allowing the present one and the next one to be stored), but this maximum can be re-defined in the compile-time options.

The ESP (Price cluster server) receives a scheduled conversion factor or calorific value from the utility company. A received value and its associated start-time are added as an entry to the relevant list on the server by the ESP application as follows:

- A new entry is added to the conversion factor list by calling the function **eSE_PriceAddConversionFactorEntry()**
- A new entry is added to the calorific value list by calling the function **eSE_PriceAddCalorificValueEntry()**

The entries are maintained in the list in increasing order of start-times. If an existing entry in the list has the same start-time as the new entry, the entry with the greater value of the Issuer Event ID is included in the list (and the other entry is discarded).

Once a new entry is added to a list on the server, a Publish Conversion Factor or Publish Calorific Value command is automatically sent to the cluster clients to inform them that a new value is available, allowing them to update their lists with the new information.

Initializing Conversion Factors and Calorific Values at Network Startup Note the following issues at network startup:

- When the ESP node first starts, there may be no other active nodes in the network to receive a new conversion factor and/or calorific value. Thus, the Price cluster clients should request this information from the ESP when they start. They can do this using **eSE_PriceGetConversionFactorSend()** or **eSE_PriceGetCalorificValueSend()**, as appropriate.
- When initializing the conversion factor or calorific value at ESP startup, the ESP application should call **eSE_PriceAddConversionFactorEntry()** or **eSE_PriceAddCalorificValueEntry()** with the address mode parameter set to E_ZCL_AM_NO_TRANSMIT. This prevents the new value from being transmitted to a network with no other active nodes.
- Any clients that are active during ESP initialization should not request a conversion factor or calorific value from the ESP before the values are received from the utility company. To avoid this problem, the ESP application should obtain the values from the utility company before calling the ZigBee PRO function **ZPS_eAplZdoStartStack()** and after calling the relevant endpoint register function (example, **eSE_RegisterEspMeterEndPoint()**).

Parent topic:*Price Cluster*

Price events The Price cluster has its own events that are handled through the callback mechanism described in *Chapter 3*. If a device uses the Price cluster then Price event handling must be included in the callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function (for example, through **eSE_RegisterEspEndPoint()** for a standalone ESP). The relevant callback function will then be invoked when a Price event occurs.

For a Price event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCus-

tomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to a tsSE_PriceCallBackMessage structure which contains the Price parameters:

```
typedef struct
{
 teSE_PriceCallBackEventType      eEventType;
 uint32              u32CurrentTime;
 union {
  tsSE_PriceTableCommand         sPriceTableCommand;
  tsSE_PriceTableTimeEvent        sPriceTableTimeEvent;
  teSE_PriceCommandOptions         ePriceCommandOptions;
  tsSE_PriceAckCmdPayload         *psAckCmdPayload;
  tsSE_PriceAttrReadInput         sReadAttrInfo;
  tsSE_BlockPeriodTableTimeEvent      sBlockPeriodTableTimeEvent;
  tsSE_ConversionFactorTableTimeEvent    sConversionFactorTableTimeEvent;
  tsSE_CalorificValueTableTimeEvent      sCalorificValueTableTimeEvent;
  } uMessage;
} tsSE_PriceCallBackMessage;
```

The eEventType field of the above structure specifies the type of Price event that has been generated - these event types are listed and described below (also refer to Section 40.12.2 for a summary of the Price events).

Note: The field sReadAttrInfo is reserved for future use (for Block mode).

E_SE_PRICE_TABLE_ADD The E_SE_PRICE_TABLE_ADD event is generated on a Price cluster client when an attempt has been made to add a scheduled price (received in a Publish Price command) to the local price list. In the tsSE_PriceCallBackMessage structure, the u32CurrentTime field is set to the current time from the Publish Price command and the sPriceTableCommand field is used as follows:

```
typedef struct {
   teSE_PriceStatus ePriceStatus;
} tsSE_PriceTableCommand;
```

ePriceStatus contains E_SE_PRICE_SUCCESS if a new price has been successfully added to the price list. Otherwise, the addition was rejected for the reason specified by ePriceStatus. If the addition was successful but the new price information overlapped (in time) any existing price information in the list, this previous price information may have been deleted from the list according to the rules in the ZigBee SE Profile specification.

E_SE_PRICE_TABLE_ACTIVE The E_SE_PRICE_TABLE_ACTIVE event is generated when there is a new active price or the active price expires. This event can occur due to a time update or the reception of a Publish Price command from the server. In the tsSE_PriceCallBackMessage structure, the u32CurrentTime field is set to the current ZCL time and the sPriceTableTimeEvent field is used as follows:

```
typedef struct {
   teSE_PriceStatus  ePriceStatus;
   uint8      u8NumberOfEntriesFree;
} tsSE_PriceTableTimeEvent;
```

ePriceStatus contains E_SE_PRICE_SUCCESS if there is a new active price or E_SE_PRICE_TABLE_NOT_YET_ACTIVE if the price at the head of the list is scheduled for a time in the future.

u8NumberOfEntriesFree contains the number of free entries in the client's price list. This number can be used to determine whether the client should issue a new Get Scheduled Prices command, in order to obtain more price entries to fill the free space in the list.

E_SE_PRICE_GET_CURRENT_PRICE_RECEIVED The E_SE_PRICE_GET_CURRENT_PRICE_RECEIVED event is generated on a Price cluster server when a Get Current Price command is received from a client. In the tsSE_PriceCallBackMessage structure, the ePriceCommandOptions field is used as follows:

```
typedef enum PACK
{
    E_SE_PRICE_REQUESTOR_RX_ON_IDLE = 0x01 // LSB set
} teSE_PriceCommandOptions;
```

This field indicates whether the client that sent the request has its radio receiver enabled when idle (e.g. sleeping), and is used as described in Section 40.5.1 and Section 40.5.2.

E_SE_PRICE_TIME_UPDATE The E_SE_PRICE_TIME_UPDATE event is generated on a Price cluster client when a Publish Price command is received from the server. In the tsSE_PriceCallBackMessage structure, the u32CurrentTime field is set to the current time from the Publish Price command. The application may then use this information to time-synchronise the device, as described in Section 40.6.

E_SE_PRICE_ACK_RECEIVED The E_SE_PRICE_ACK_RECEIVED event is generated on a Price cluster server when a Price Acknowledgment command is received from a client. In the tsSE_PriceCallBackMessage structure, the psAckCmdPayload field is a pointer to the structure tsSE_PriceAckCmdPayload defined as follows:

```
typedef struct {
   uint32          u32ProviderId;
   uint32          u32IssuerEventId;
   uint32          u32PriceAckTime;
   uint8           u8Control;
} tsSE_PriceAckCmdPayload;
```

This structure contains the Price Acknowledgement command payload.

E_SE_PRICE_NO_PRICE_TABLES The E_SE_PRICE_NO_PRICE_TABLES event is generated when an active price expires, is deleted from the price list and the price list becomes empty. In the tsSE_PriceCallBackMessage structure the sPriceTableTimeEvent field is used as follows:

```
typedef struct {
   teSE_PriceStatus ePriceStatus;
   uint8 u8NumberOfEntriesFree;
} tsSE_PriceTableTimeEvent;
```

ePriceStatus contains E_SE_PRICE_NO_TABLES.

u8NumberOfEntriesFree contains the number of free entries in the client's price list. This number can be used to determine whether the client should issue a new Get Scheduled Prices command, in order to obtain more price entries to fill the free space in the list.

E_SE_PRICE_CONVERSION_FACTOR_TABLE_ACTIVE The E_SE_PRICE_CONVERSION_FACTOR_TABLE_ACTIV event is generated when a new conversion factor value becomes active - that is, when the start-time of the conversion factor entry becomes less than or equal to the present time. This event can occur due to a time update or the reception of a Publish Conversion Factor command from the server.

In the tsSE_PriceCallBackMessage structure, the u32CurrentTime field is set to the current ZCL time and the field tsSE_PriceConversionFactorTableTimeEvent is used as follows:

```
typedef struct {
   teSE_PriceStatus      eConversionFactorStatus;
   uint8          u8NumberOfEntriesFree;
} tsSE_ConversionFactorTableTimeEvent;
```

eConversionFactorStatus takes the value E_ZCL_SUCCESS when a new conversion factor becomes active.

u8NumberOfEntriesFree contains the present number of free entries in the conversion factor list. This value should be checked by the client before issuing a Get Conversion Factor command to obtain a new conversion factor value - the command should be issued only if there is free space in the list for a new entry to be added.

E_SE_PRICE_CONVERSION_FACTOR_ADD The E_SE_PRICE_CONVERSION_FACTOR_ADD event is generated when a new conversion factor entry is advertised by the ESP to the client application using the Publish Conversion Factor command. Note that the event is generated even when the new entry is not successfully added to the internal conversion factor list maintained by the cluster.

The status of the command is passed back to the user application in the ePriceStatus field of the tsSE_PriceTableCommand structure (see above) within the tsSE_PriceCallBackMessage structure.

E_SE_PRICE_CALORIFIC_VALUE_TABLE_ACTIVE The E_SE_PRICE_CALORIFIC_VALUE_TABLE_ACTIVE event is generated when a new calorific value becomes active - that is, when the start-time of the calorific value entry becomes less than or equal to the present time. This event can occur due to a time update or the reception of a Publish Calorific Value command from the server.

In the tsSE_PriceCallBackMessage structure, the u32CurrentTime field is set to the current ZCL time and the field tsSE_PriceCalorificValueTableTimeEvent is used as follows:

```
typedef struct {
   teSE_PriceStatus    eCalorificValueStatus;
   uint8        u8NumberOfEntriesFree;
} tsSE_CalorificValueTableTimeEvent;
```

eCalorificValueStatus takes the value E_ZCL_SUCCESS when a new calorific value becomes active.

u8NumberOfEntriesFree contains the present number of free entries in the calorific value list. This value should be checked by the client before issuing a Get Calorific Value command to obtain a new calorific value - the command should be issued only if there is free space in the list for a new entry to be added.

E_SE_PRICE_CALORIFIC_VALUE_ADD The E_SE_PRICE_CALORIFIC_VALUE_ADD event is generated when a new calorific value entry is advertised by the ESP to the client application using the Publish Calorific Value command. Note that this event is generated even when the new entry is not successfully added to the internal calorific value list maintained by the cluster.

The status of the command is passed back to the user application in the ePriceStatus field of the tsSE_PriceTableCommand structure (see above) within the tsSE_PriceCallBackMessage structure.

Parent topic:*Price Cluster*

Functions The following Price cluster functions are provided:

- eSE_PriceCreate
- eSE_PriceGetCurrentPriceSend
- eSE_PriceGetScheduledPricesSend
- eSE_PriceAddPriceEntry
- eSE_PriceAddPriceEntryToClient
- eSE_PriceGetPriceEntry
- eSE_PriceDoesPriceEntryExist
- eSE_PriceRemovePriceEntry
- eSE_PriceClearAllPriceEntries
- eSE_PriceAddConversionFactorEntry
- eSE_PriceGetConversionFactorSend
- eSE_PriceGetConversionFactorEntry
- eSE_PriceDoesConversionFactorEntryExist
- eSE_PriceRemoveConversionFactorEntry
- eSE_PriceClearAllConversionFactorEntries
- eSE_PriceAddCalorificValueEntry
- eSE_PriceGetCalorificValueSend
- eSE_PriceGetCalorificValueEntry
- eSE_PriceDoesCalorificValueEntryExist
- eSE_PriceRemoveCalorificValueEntry
- eSE_PriceClearAllCalorificValueEntries

eSE_PriceCreate

```
teSE_PriceStatus eSE_PriceCreate(
        bool_t bIsServer,
        uint8 u8NumberOfRecordEntries,
        uint8 *pu8AttributeControlBits,
        uint8 *pau8RateLabel,
        tsZCL_ClusterInstance *psClusterInstance,
        tsZCL_ClusterDefinition *psClusterDefinition,
        tsSE_PriceCustomDataStructure
        *psCustomDataStructure,
        tsSE_PricePublishPriceRecord
        *psPublishPriceRecord,
        void *pvEndPointSharedStructPtr);
```

Description This function creates an instance of the Price cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device).

This function will create a Price cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to *Appendix D.*

Note: This function must not be called for an endpoint on which a standard ZigBee device (e.g. IPD) will be used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in the *ZigBee Devices User Guide (JNUG3131)*

When used, this function must be the first Price cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate be the total number of attributes supported by the Price cluster, which can be obtained by using the macro PRICE_NUM_OF_ATTRIBUTES.

The array declaration should be as follows:

```
uint8 au8AppPriceClusterAttributeControlBits[PRICE_NUM_OF_ATTRIBUTES];
```

The function initializes the array elements to zero.

The function also requires an array of price labels to be declared, in which each array element is a label (string) for each price in the price list. The required declarations are different for a cluster server and client, as follows:

```
uint8 au8RateLabel[SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES][SE_PRICE_
↪CLIENT_MAX_STRING_LENGTH];
uint8 au8RateLabel[SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES][SE_PRICE_
↪CLIENT_MAX_STRING_LENGTH];
```

Parameters

- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *u8NumberOfRecordEntries* Number of prices that can be stored in the price list, one of:
- SE_PRICE_NUMBER_OF_SERVER_PRICE_RECORD_ENTRIES
- SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
- *pau8RateLabel* Pointer to an array of price labels (strings), with one element for each price in the price list (see above).
- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Price cluster. This parameter can refer to a pre-filled structure called sCLD_Price which is provided in the **Price.h** file.
- *psCustomDataStructure* Pointer to structure which contains custom data for the Price cluster. This structure is used for internal data storage. No knowledge of the fields of this structure is required

- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_Price which defines the attributes of Price cluster. The function initializes the attributes with default values.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eSE_PriceGetCurrentPriceSend

```
teZCL_Status eSE_PriceGetCurrentPriceSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    teSE_PriceCommandOptions ePriceCommandOptions);
```

Description This function can be used on a Price cluster client to send a Get Current Price command to the Price cluster server. Therefore, it is used by a device (such as an IPD) to obtain the currently active price from the ESP.

The ESP should respond with a Publish Price command containing the active price. This response is processed by the Price cluster. The obtained price is checked against the prices currently in the price list on the client. If the price is not currently in the list, it is added to the list and an E_SE_PRICE_TABLE_ADD event is generated to indicate that a price has been added.

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId* Number of the remote endpoint to which the request is sent
- *psDestinationAddress* Pointer to a structure containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *ePriceCommandOptions* Indicates whether the radio receiver on client remains on when the device is idle (for example, asleep):
  - 0x01 - receiver on when idle
  - 0x00 - receiver off when idle
- An enumeration is provided for the ‘on’ case:
- E_SE_PRICE_REQUESTOR_RX_ON_IDLE

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL

Parent topic: Functions

eSE_PriceGetScheduledPricesSend

```
teZCL_Status eSE_PriceGetScheduledPricesSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint32 u32StartTime,
        int8 u8NumberOfEvents);
```

Description This function can be used on a Price cluster client to send a Get Scheduled Prices command to the Price cluster server. Therefore, it is used by a device (such as an IPD) to obtain the current price schedule from the ESP, either to check that its own price schedule is up-to-date or to recover the price schedule following a device reset.

You must specify the earliest start-time for the scheduled prices to be included in the results. This is normally set to zero or the current time (UTC). Note that you are not advised to specify the last time in the client price list, since the server may contain updates for prices covering an earlier time-period that are already in the client price list. You must also specify the maximum number of scheduled prices to be returned in the results.

The ESP should respond with multiple Publish Price commands containing the scheduled prices. Each response is processed by the Price cluster. The obtained price is checked against the prices currently in the price list on the client. If the price is not currently in the list, it is added to the list and an E_SE_PRICE_TABLE_ADD event is generated to indicate that a price has been added.

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId* Number of the remote endpoint to which the request is sent
- *psDestinationAddress* Pointer to a structure containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u32StartTime* The earliest start-time of any prices to be returned - this is normally set to zero or the current time (UTC)
- *u8NumberOfEvents* The maximum number of scheduled prices to be returned in the results - this should normally be set to: SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_ZBUFFER_FAIL

Parent topic: Functions

eSE_PriceAddPriceEntry

```
teSE_PriceStatus eSE_PriceAddPriceEntry(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    bool_t bOverwritePrevious,
    tsSE_PricePublishPriceCmdPayload *psPricePayload,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on the Price cluster server to add a price to the local price list. The function also sends an unsolicited Publish Price command containing the new price information to one or more remote endpoints. The function should be called on the ESP when a new price is received from the utility company.

On receiving the Publish Price command, a remote client will automatically add the new price to the local price list. However, you must specify the action to be taken if the time-period of the new price overlaps with the time-period of a price that is already in the client's price list. You can choose to delete the existing price and add the new price, or leave the existing price in place and not add the new price. The rules on overlapping prices are defined in the ZigBee Smart Energy Profile specification.

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId* Number of the remote endpoint to which the request is sent
- *psDestinationAddress* Pointer to a structure containing the address of the remote node to which the Publish Price command is sent. It is recommended that the command is sent to all bound clients using a ZCL address mode of E_ZCL_AM_BOUND. If the stack has not been started, the E_ZCL_AM_NO_TRANSMIT address mode should be used
- *bOverwritePrevious* Action to be taken if the new price overlaps (in time) a price which is already in the price list:
  - TRUE - existing price deleted, new price added
  - FALSE - new price not added and error returned
- *psPricePayload* Pointer to a structure containing the price information to be added (see Section 40.11.1). This parameter only needs to remain in scope for the duration of this function call
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the command

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE
- E_ZCL_ERR_TIME_NOT_SYNCHRONISED
- E_ZCL_ERR_INSUFFICIENT_SPACE
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_SE_PRICE_OVERFLOW
- E_SE_PRICE_DUPLICATE
- E_SE_PRICE_DATA_OLD

Parent topic: Functions

eSE_PriceAddPriceEntryToClient

```
teSE_PriceStatus eSE_PriceAddPriceEntryToClient(
    uint8 u8SourceEndPointId,
    bool_t bOverwritePrevious,
    tsSE_PricePublishPriceCmdPayload *psPricePayload);
```

Description This function can be used on a Price cluster client to add a price to the local price list directly.

Normally, price entries are automatically added to the price list on a client when a Publish Price command is received from the server (e.g. the ESP). However, this function can be used by the local application to directly add a price entry to the price list on the client. The function should therefore only be used on a device which does not receive price information from the server (but by some other means, such as via the Internet).

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which the request is sent
- *bOverwritePrevious* Action to be taken if the new price overlaps (in time) a price which is already in the price list:
  - TRUE - existing price deleted, new price added
  - FALSE - new price not added and error returned
- *psPricePayload* Pointer to a structure containing the price information to be added (see Section 40.11.1). This parameter only needs to remain in scope for the duration of this function call

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE
- E_ZCL_ERR_TIME_NOT_SYNCHRONISED
- E_ZCL_ERR_INSUFFICIENT_SPACE
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_SE_PRICE_OVERFLOW
- E_SE_PRICE_DUPLICATE
- E_SE_PRICE_DATA_OLD

Parent topic: Functions

eSE_PriceGetPriceEntry

```
teSE_PriceStatus eSE_PriceGetPriceEntry(
    uint8 u8SourceEndPointId,
    bool_t bIsServer,
    uint8 u8TableIndex,
    tsSE_PricePublishPriceCmdPayload **psPricePayload);
```

Description This function can be used to obtain the entry with specified index from a price list on the local device. For example, the function can be used on an IPD to obtain a price to display.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

Parameters

- *u8SourceEndPointId* Number of the local endpoint for the price list to be accessed
- *bIsServer* Nature of the Price cluster instance containing the price list:
  - TRUE - server (for example, on ESP)
  - FALSE - client (for example, on IPD)
- *u8TableIndex* The index of the price entry to obtain from the price list (index 0 is the entry with the oldest start-time and may contain the currently active price)
- *psPricePayload* Pointer to a pointer to a structure which will be used to store the obtained price information (see Section 40.11.1), if found. The pointer value that is returned in this parameter points to the structure in the internal storage associated with the list. The data in the structure will be valid as long as the item remains in the list

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_SE_PRICE_TABLE_NOT_FOUND

Parent topic: Functions

eSE_PriceDoesPriceEntryExist

```
teSE_PriceStatus eSE_PriceDoesPriceEntryExist(
    uint8 u8SourceEndPointId,
    bool_t bIsServer,
    uint32 u32StartTime);
```

Description This function can be used to check whether a price entry with the specified start-time is present in a price list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

For a price entry to be successfully found, the specified start-time must exactly match the start-time of an entry in the price list, otherwise the status code E_SE_PRICE_NOT_FOUND will be returned.

Parameters

- *u8SourceEndPointId* Number of the local endpoint for the price list to be accessed
- *bIsServer* Nature of the Price cluster instance containing the price list:
  - TRUE - server
  - FALSE - client
- *u32StartTime* Start-time of the price entry to search for

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_SE_PRICE_NOT_FOUND

Parent topic: Functions

eSE_PriceRemovePriceEntry

```
teSE_PriceStatus eSE_PriceRemovePriceEntry(
    uint8 u8SourceEndPointId,
    bool_t bIsServer,
    uint32 u32StartTime);
```

Description This function can be used to delete a price entry with specified start-time from a price list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

For the successful deletion of a price entry, the specified start-time must exactly match the start-time of an entry in the price list, otherwise the status code E_SE_PRICE_NOT_FOUND will be returned.

Parameters

- *u8SourceEndPointId* Number of the local endpoint on which Price cluster resides
- *bIsServer* Nature of the Price cluster instance containing the price list:
  - TRUE - server
  - FALSE - client
- *u32StartTime* The start-time of the price entry to delete

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_SE_PRICE_NOT_FOUND
- E_SE_PRICE_TABLE_NOT_FOUND

Parent topic: Functions

eSE_PriceClearAllPriceEntries

```
teSE_PriceStatus eSE_PriceClearAllPriceEntries(
    uint8 u8SourceEndPointId,
    bool_t bIsServer);
```

Description This function can be used to delete all entries in a price list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

Parameters

- *u8SourceEndPointId* Number of the local endpoint for the price list to be cleared
- *bIsServer* Nature of the Price cluster instance containing the price list:
  - TRUE - server
  - FALSE - client

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND

Parent topic: Functions

eSE_PriceAddConversionFactorEntry

```
teZCL_Status eSE_PriceAddConversionFactorEntry(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    bool_t bOverwritePrevious,
    tsSE_PricePublishConversionCmdPayload
```

(continues on next page)

(continued from previous page)

```
*psPublishConversionCmdPayload,
uint8 *pu8TransactionSequenceNumber);
```

Description The function can be used on a Price cluster server to add a new conversion factor entry to the internal list of scheduled conversion factors maintained by the cluster. The function also sends an unsolicited Publish Conversion Factor command to the Price cluster client nodes in the network, to advertise the new conversion factor. Therefore, the function should be called on the ESP when a new conversion factor is received from the utility company.

On receiving the Publish Conversion Factor command, a remote client automatically adds the new conversion factor to the local conversion factor list. However, if the new entry has the same start-time as an existing entry in the list, the outcome depends on the setting of the boolean parameter *bOverwritePrevious* in this function:

- If this parameter is set to TRUE then the existing entry is removed and the new entry is added
- If this parameter is set to FALSE then the Issuer Event IDs of the two conversion factor entries are compared:
  - If the Event ID of the new entry is the greater, the existing entry is removed and the new entry is added
  - If the Event ID of the existing entry is the greater, E_ZCL_FAIL is returned and the list is not modified

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId* Number of the remote endpoint to which the request is sent
- *psDestinationAddress* Pointer to a structure containing the address of the remote node to which the request is sent
- *bOverwritePrevious* Determines whether an existing conversion factor with the same start-time on the clients will be over-written without comparing Event IDs (see above):
  - TRUE - over-write existing entry
  - FALSE - compare Event IDs first
- *psPublishConversionCmdPayload* Pointer to conversion factor entry to be added to list on server and advertised to clients
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_TIME_NOT_SYNCHRONISED

Parent topic: Functions

eSE_PriceGetConversionFactorSend

```
teZCL_Status eSE_PriceGetConversionFactorSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint32 u32StartTime,
        uint8 u8NumberOfEvents);
```

Description The function can be used on a Price cluster client to send a Get Conversion Factor request to the Price cluster server. Therefore, it is used by a device (such as an IPD) to obtain scheduled conversion factor values from the ESP. The function allows scheduled conversion factors to be obtained with start-times greater than or equal to a specified time, *u32StartTime*.

The ESP should respond with a Publish Conversion Factor command containing up to *u8NumberOfEvent* scheduled conversion factor values. The Price cluster on the receiving client processes the response by updating the local conversion factor list, as follows. For each conversion factor received in the response, the event E_SE_PRICE_CONVERSION_FACTOR_ADD is generated.

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId* Number of the remote endpoint to which the request is sent
- *psDestinationAddress* Pointer to a structure containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u32StartTime* Earliest start-time of scheduled conversion factors to be returned - a setting of 0 returns the factor that is currently active and factors with start-times in the future
- *u8NumberOfEvents* Maximum number of conversion factors to be returned as a result of this request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Functions

eSE_PriceGetConversionFactorEntry

```
teSE_PriceStatus eSE_PriceGetConversionFactorEntry(
    uint8 u8SourceEndPointId,
    bool_t bIsServer,
    uint8 u8TableIndex,
    sSE_PricePublishConversionCmdPayload
    **ppsPublishConversionCmdPayload);
```

Description This function can be used to obtain the entry with the specified index from the conversion factor list on the local device. For example, the function can be used on an IPD to obtain a conversion factor to display.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint for the conversion factor list to be accessed
- *bIsServer*: Nature of the Price cluster instance containing the list:
  - TRUE - server (example on ESP)
  - FALSE - client (example on IPD)
- *u8TableIndex*: The index of the entry to obtain from the conversion factor list (index 0 is the entry with the oldest start-time and may contain the currently active conversion factor)
- *ppsPublishConversionCmdPayload*: Pointer to a pointer to a structure which will be used to store the obtained conversion factor information (see Section40.11.2), if found. The pointer value that is returned in this parameter points to the structure in the internal storage associated with the list. The data in the structure will be valid as long as the item remains in the list

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_SE_PRICE_TABLE_NOT_FOUND

Parent topic: Functions

eSE_PriceDoesConversionFactorEntryExist

```
teSE_PriceStatus eSE_PriceDoesConversionFactorEntryExist(
    uint8 u8SourceEndPointId,
    bool_t bIsServer,
    uint32 u32StartTime);
```

Description This function can be used to check whether a conversion factor entry with the specified start-time is present in a conversion factor list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

For a conversion factor entry to be successfully found, the specified start-time must exactly match the start-time of an entry in the conversion factor list, otherwise the status code E_SE_PRICE_NOT_FOUND will be returned.

Parameters

- *u8SourceEndPointId* Number of the local endpoint for the conversion factor list to be accessed
- *bIsServer* Nature of the Price cluster instance containing the price list:
- TRUE - server
- FALSE - client
- *u32StartTime* Start-time of the conversion factor entry to search for

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_SE_PRICE_NOT_FOUND

Parent topic: Functions

eSE_PriceRemoveConversionFactorEntry

```
teSE_PriceStatus eSE_PriceRemoveConversionFactorEntry(
    uint8 u8SourceEndPointId,
    bool_t bIsServer,
    uint32 u32StartTime);
```

Description This function can be used to delete a conversion factor entry with specified start-time from conversion factor list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

For the successful deletion of a conversion factor entry, the specified start-time must exactly match the start-time of an entry in the conversion factor list, otherwise the status code E_SE_PRICE_NOT_FOUND is returned.

Parameters

- *u8SourceEndPointId* Number of the local endpoint for the conversion factor list to be accessed
- *bIsServer* Nature of the Price cluster instance containing the list:
  - TRUE - server
  - FALSE - client
- *u32StartTime* The start-time of the conversion factor entry to delete

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_SE_PRICE_NOT_FOUND
- E_SE_PRICE_TABLE_NOT_FOUND

Parent topic: Functions

eSE_PriceClearAllConversionFactorEntries

```
teSE_PriceStatus eSE_PriceClearAllConversionFactorEntries(
    uint8 u8SourceEndPointId,
    bool_t bIsServer);
```

Description This function can be used to delete all entries in a conversion factor list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

Parameters

- *u8SourceEndPointId* Number of the local endpoint for the conversion factor list to be cleared
- *bIsServer* Nature of the Price cluster instance containing the price list:
- TRUE - server
- FALSE - client

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND

Parent topic: Functions

eSE_PriceAddCalorificValueEntry

```
teZCL_Status eSE_PriceAddCalorificValueEntry(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    bool_t bOverwritePrevious,
    tsSE_PricePublishCalorificValueCmdPayload
    *psPublishCalorificValueCmdPayload,
    uint8 *pu8TransactionSequenceNumber);
```

Description The function can be used on a Price cluster server to add a calorific value entry to the internal list of scheduled calorific values maintained by the cluster. The function also sends an unsolicited Publish Calorific Value command to the Price cluster client nodes in the network, to advertise the new calorific value. Therefore, the function should be called on the ESP when a new calorific value is received from the utility company.

On receiving the Publish Calorific Value command, a remote client automatically adds the new calorific value to the local calorific value list. However, if the new entry has the same start-time as an existing entry in the list, the outcome depends on the setting of the boolean parameter *bOverwritePrevious* in this function:

- If this parameter is set to TRUE then the existing entry is removed and the new entry is added
- If this parameter is set to FALSE then the Issuer Event IDs of the two calorific value entries are compared:
  - If the Event ID of the new entry is the greater, the existing entry is removed and the new entry is added
  - If the Event ID of the existing entry is the greater, E_ZCL_FAIL is returned and the list is not modified

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent
- *psDestinationAddress*: Pointer to a structure containing the address of the remote node to which the request is sent
- *bOverwritePrevious*: Determines whether an existing calorific value with the same start-time on the clients will be over-written without comparing Event IDs (see above):
  - TRUE - over-write existing entry
  - FALSE - compare Event IDs first
- *psPublishCalorificValueCmdPayload* Pointer to calorific value entry to be added to list on server and advertised to clients
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_TIME_NOT_SYNCHRONISED

Parent topic: Functions

eSE_PriceGetCalorificValueSend

```
teZCL_Status eSE_PriceGetCalorificValueSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
```

(continues on next page)

(continued from previous page)

```
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint32 u32StartTime,
    uint8 u8NumberOfEvents);
```

Description The function can be used on a Price cluster client to send a Get Calorific Value request to the Price cluster server. Therefore, it is used by a device (such as an IPD) to obtain scheduled calorific values from the ESP. The function allows scheduled calorific values to be obtained with start-times greater than or equal to a specified time, *u32StartTime*.

The ESP should respond with a Publish Calorific Value command containing up to *u8NumberOfEvent* scheduled calorific values. The Price cluster on the receiving client processes the response by updating the local calorific value list, as follows. For each calorific value received in the response, the event E_SE_PRICE_CALORIFIC_VALUE_ADD is generated.

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent
- *psDestinationAddress*: Pointer to a structure containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u32StartTime*: Earliest start-time of scheduled calorific values to be returned - a setting of 0 returns the value that is currently active and values with start-times in the future
- *u8NumberOfEvents*: Maximum number of calorific values to be returned as a result of this request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZTRANSMIT_FAIL

Parent topic: Functions

eSE_PriceGetCalorificValueEntry

```
teSE_PriceStatus eSE_PriceGetCalorificValueEntry(
    uint8 u8SourceEndPointId,
    bool_t bIsServer,
    uint8 u8TableIndex,
    sSE_PricePublishCalorificValueCmdPayload
    **ppsPublishCalorificValueCmdPayload);
```

Description This function can be used to obtain the entry with the specified index from the calorific value list on the local device. For example, the function can be used on an IPD to obtain a calorific value to display.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

Parameters

- *u8SourceEndPointId* Number of the local endpoint for the calorific value list to be accessed
- *bIsServer* Nature of the Price cluster instance containing the list:
  - TRUE - server (example on ESP)
  - FALSE - client (example on IPD)
- *u8TableIndex* The index of the entry to obtain from the calorific value list (index 0 is the entry with the oldest start-time and may contain the currently active calorific value)
- **ppsPublishCalorificValueCmdPayload: Pointer to a pointer to a structure which will be used to store the obtained calorific value information (see Section 40.11.3), if found. The pointer value that is returned in this parameter points to the structure in the internal storage associated with the list. The data in the structure will be valid as long as the item remains in the list

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_SE_PRICE_TABLE_NOT_FOUND

Parent topic: Functions

eSE_PriceDoesCalorificValueEntryExist

```
teSE_PriceStatus eSE_PriceDoesCalorificValueEntryExist(
    uint8 u8SourceEndPointId,
    bool_t bIsServer,
    uint32 u32StartTime);
```

Description This function can be used to check whether a calorific value entry with the specified start-time is present in a calorific value list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

For a calorific value entry to be successfully found, the specified start-time must exactly match the start-time of an entry in the calorific value list, otherwise the status code E_SE_PRICE_NOT_FOUND will be returned.

Parameters

- *u8SourceEndPointId* Number of the local endpoint for the calorific value list to be accessed
- *bIsServer* Nature of the Price cluster instance containing the price list:
  - TRUE - server
  - FALSE - client
- *u32StartTime* Start-time of the calorific value entry to search for

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_SE_PRICE_NOT_FOUND

Parent topic: Functions

eSE_PriceRemoveCalorificValueEntry

```
teSE_PriceStatus eSE_PriceRemoveCalorificValueEntry(
    uint8 u8SourceEndPointId,
    bool_t bIsServer,
    uint32 u32StartTime);
```

Description This function can be used to delete a calorific value entry with specified start-time from calorific value list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

For the successful deletion of a calorific value entry, the specified start-time must exactly match the start-time of an entry in the calorific value list, otherwise the status code E_SE_PRICE_NOT_FOUND will be returned.

Parameters

- *u8SourceEndPointId* Number of the local endpoint for the calorific value list to be accessed
- *bIsServer* Nature of the Price cluster instance containing the list:
- TRUE - server
- FALSE - client
- *u32StartTime* Start-time of the calorific value entry to delete

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_SE_PRICE_NOT_FOUND
- E_SE_PRICE_TABLE_NOT_FOUND

Parent topic: Functions

eSE_PriceClearAllCalorificValueEntries

```
teSE_PriceStatus eSE_PriceClearAllCalorificValueEntries(
    uint8 u8SourceEndPointId,
    bool_t bIsServer);
```

Description This function can be used to delete all entries in a calorific value list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

Parameters

- *u8SourceEndPointId* Number of the local endpoint for the calorific value list to be cleared
- *bIsServer* Nature of the Price cluster instance containing the price list:
  - TRUE - server
  - FALSE - client

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND

Parent topic: Functions

Parent topic:*Price Cluster*

Return codes In addition to some of the ZCL status enumerations, the following enumerations are returned by Price cluster functions (see Section 40.9) to indicate the outcome of the function call.

```
typedef enum PACK
{
    E_SE_PRICE_OVERLAP =0x80,
    E_SE_PRICE_TABLE_NOT_YET_ACTIVE,
    E_SE_PRICE_DATA_OLD,
    E_SE_PRICE_NOT_FOUND,
    E_SE_PRICE_TABLE_NOT_FOUND,
    E_SE_PRICE_OVERFLOW,
    E_SE_PRICE_DUPLICATE,
    E_SE_PRICE_NO_TABLES,
    E_SE_PRICE_BLOCK_PERIOD_TABLE_NOT_YET_ACTIVE,
    E_SE_PRICE_NO_BLOCKS,
```

(continues on next page)

(continued from previous page)

```
    E_SE_PRICE_NUMBER_OF_BLOCK_THRESHOLD_MISMATCH,
    E_SE_BLOCK_PERIOD_OVERFLOW,
    E_SE_BLOCK_PERIOD_DUPLICATE,
    E_SE_BLOCK_PERIOD_DATA_OLD,
    E_SE_BLOCK_PERIOD_OVERLAP,
    E_SE_PRICE_STATUS_ENUM_END
} teSE_PriceStatus;
```

The above enumerations are described in the table below.

Enumeration	Description
E_SE_PRICE_OVERLAP	New price overlaps (in time) with existing price in price list
E_SE_PRICE_TABLE_NOT_	No active price at head of price list
E_SE_PRICE_DATA_OLD	Attempt made to add price which overlaps (in time) with existing price in price list and which is older than existing price *
E_SE_PRICE_NOT_FOUND	Specified price was not found in price list
E_SE_PRICE_TABLE_NOT_	Specified price list was not found
E_SE_PRICE_OVERFLOW	Attempt to add price to price list failed because end-time for new price (start-time + duration x 60) exceeds maximum permissible time value of 0xFFFFFFFFF (UTC)
E_SE_PRICE_DUPLICATE	Specified price information already exists in price list
E_SE_PRICE_NO_TABLES	Reserved for future use (for Block mode)
E_SE_PRICE_BLOCK_PERI BLE_NOT_YET_ACTIVE	Reserved for future use (for Block mode)
E_SE_PRICE_NO_BLOCKS	Reserved for future use (for Block mode)
E_SE_PRICE_NUMBER_OF	Reserved for future use (for Block mode)
E_SE_BLOCK_PERIOD_OV	Reserved for future use (for Block mode)
E_SE_BLOCK_PERIOD_DU	Reserved for future use (for Block mode)
E_SE_BLOCK_PERIOD_DA	Reserved for future use (for Block mode)
E_SE_BLOCK_PERIOD_OV	Reserved for future use (for Block mode)

* Value of u32IssuerEventId in tsSE_PricePublishPriceCmdPayload structure (see Section 40.11.1) is less for the price to be added than for the existing (overlapping) price.

Parent topic:*Price Cluster*

Structures

tsSE_PricePublishPriceCmdPayload This structure is used to hold price information to be added to a price list of a Price cluster:

```
typedef struct {
    uint8       u8UnitOfMeasure;
    uint8       u8PriceTrailingDigitAndPriceTier;
    uint8       u8NumberOfPriceTiersAndRegisterTiers;
    uint8       u8PriceRatio;
    uint8       u8GenerationPriceRatio;
    uint8       u8AlternateCostUnit;
    uint8       u8AlternateCostTrailingDigit;
    uint8       u8NumberOfBlockThresholds;
    uint8       u8PriceControl;
    uint16      u16Currency;
    uint16      u16DurationInMinutes;
    uint32      u32ProviderId;
```

(continues on next page)

(continued from previous page)

```
    uint32              u32IssuerEventId;
    uint32              u32StartTime;
    uint32              u32Price;
    uint32              u32GenerationPrice;
    uint32              u32AlternateCostDelivered;
    tsZCL_OctetString   sRateLabel;
} tsSE_PricePublishPriceCmdPayload;
```

where:

- u8UnitOfMeasure indicates the resource (e.g. electricity) and unit of measure (e.g. kWh) for the pricing (see Section 42.10.3)
- u8PriceTrailingDigitAndPriceTier is an 8-bit bitmap indicating the price tier and the number of digits after the decimal point in the price:
  - The 4 most significant bits give the number of digits to the right of the decimal point in the price
  - The 4 least significant bits give the price tier in the range 1 to 6
- u8NumberOfPriceTiersAndRegisterTiers is an 8-bit bitmap indicating the number of price tiers available and the particular tier that the price information in the structure relates to:
  - The 4 most significant bits give the number of available price tiers in the range 0 to 6
  - The 4 least significant bits give the price tier used in the range 1 to 6

    (this value must be less than or equal to the value in the 4 leading bits)
- u8PriceRatio (optional) is the ratio of the price quoted in u32Price to the 'normal' price offered by the utility company. The actual price ratio should be multiplied by 10 for encoding this field, so that a field value of 0x01 represents 0.1 and 0xFE represents 25.4, while 0xFF indicates that the field is not used
- u8GenerationPriceRatio (optional) is the ratio of the price quoted in u32GenerationPrice to the 'normal' price offered by the utility company. The actual price ratio should be multiplied by 10 for encoding this field, so that a field value of 0x01 represents 0.1 and 0xFE represents 25.4, while 0xFF is reserved to indicate that the field is not used
- u8AlternateCostUnit (optional) is an 8-bit bitmap indicating the unit for the alternative cost in u32AlternateCostDelivered. Currently, the only supported unit is kilograms of CO2, indicated by the value 0x01
- u8AlternateCostTrailingDigit (optional) is an 8-bit bitmap in which the 4 most significant bits indicate the number of digits after the decimal point in u32AlternateCostDelivered (the 4 least significant bits are reserved)
- u8NumberOfBlockThresholds is reserved for future use (for Block mode)
- u8PriceControl is reserved for future use (for Block mode)
- u16Currency indicates the currency (e.g. Euro) used for the price - this field should be set to the appropriate value defined by ISO 4217
- u16DurationInMinutes indicates the duration, in minutes, for which the price will be valid (0xFFFF indicates that price will remain valid until changed)
- u32ProviderId is an identifier for the utility company
- u``32IssuerEventId is a unique identifier for the price information - the higher its value, the more recently the price information was issued (a UTC time-stamp could be used in this field)
- u``32StartTime indicates the start-time (UTC) for the price, in seconds. The special value 0x00000000 denotes a start-time of 'now'

- u``32Price is the resource price per unit indicated in u8UnitOfMeasure, expressed in the currency indicated in u16Currency, with the position of the decimal point as indicated in u8PriceTrailingDigitAndPriceTie``r
- u``32GenerationPrice (optional) is the resource price per unit indicated in u8UnitOfMeasure, expressed in the currency indicated in u16Currency and with the position of the decimal point as indicated in u8PriceTrailingDigitAndPriceTier, for a resource that is generated on the customer premises and supplied to the utility company (e.g. solar-sourced electric power supplied to the national grid). A value of 0xFFFFFFFF indicates that this field is not used
- u32AlternateCostDelivered (optional) indicates an alternative cost (per resource consumption unit) which is measured by a means other than monetary - for example, the amount of CO2 emitted per unit of gas consumed This alternative cost is interpreted as specified by u8AlternateCostUnit and u8AlternateCostTrailingDigit
- sRateLabel is a string of up to 12 characters containing a label for the price information in the structure

Parent topic: Structures

tsSE_PricePublishConversionCmdPayload This structure is used to hold information to be added to a conversion factor list of a Price cluster:

```
typedef struct {
    uint32          u32IssuerEventId;
    uint32          u32StartTime;
    uint32          u32ConversionFactor;
    zbmap8            u8ConversionFactorTrailingDigit;
}tsSE_PricePublishConversionCmdPayload;
```

where:

- u32IssuerEventId is a unique identifier for the conversion factor information - the higher the value, the more recently the information was issued
- u32StartTime is the start-time of the conversion factor value. This is the time at which the conversion factor value is scheduled to become active
- u32ConversionFactor is used only for gas and accounts for the variation in the volume of gas with temperature and pressure (the value is dimensionless)
- u8ConversionFactorTrailingDigit is an 8-bit bitmap which indicates the location of the decimal point in the u32ConversionFactor field. The most significant 4 bits indicate the number of digits after the decimal point. The remaining bits are reserved

Parent topic: Structures

tsSE_PricePublishCalorificValueCmdPayload This structure is used to hold information to be added to a calorific value list of the Price cluster:

```
typedef struct {
    zenum8                u8CalorificValueUnit;
    zbmap8                u8CalorificValueTrailingDigit;
    uint32               u32IssuerEventId;
    uint32               u32StartTime;
    uint32               u32CalorificValue;
}tsSE_PricePublishCalorificValueCmdPayload;
```

where:

- u8CalorificValueUnit is an 8-bit enumerated value which defines the unit for the u32CalorificValue field (below). It indicates whether the calorific value is quantified per unit volume or per unit mass - see Section 40.12.3.
- u8CalorificValueTrailingDigit is an 8-bit bitmap which indicates the location of the decimal point in the u32CalorificValue field (below). The most significant 4 bits indicate the number of digits after the decimal point. The remaining bits are reserved
- u32IssuerEventId is a unique identifier for the calorific value information - the higher the value, the more recently the information was issued
- u32StartTime is the start-time of the calorific value. This is the time at which the conversion factor value is scheduled to become active
- u32CalorificValue is used only for gas and indicates the quantity of energy in MJ that is generated per unit volume or unit mass of gas burned (see u8CalorificValueUnit). The position of the decimal point is indicated by u8CalorificValueTrailingDigit described above

Parent topic: Structures

Parent topic:*Price Cluster*

Enumerations

'Attribute ID' Enumerations The following structure contains the enumerations used to identify the attributes of the Price cluster.

Note: Only the Tier Label attributes are currently used. The remaining attributes are reserved for future use (for Block mode).

```
typedef enum PACK
{
/*Price Cluster Attribute Tier Price Label Set Attr Ids (D.4.2.2.1)*/
    E_CLD_P_ATTR_TIER_1_PRICE_LABEL = 0x0000,
    E_CLD_P_ATTR_TIER_2_PRICE_LABEL,
    ...
    ...
    E_CLD_P_ATTR_TIER_15_PRICE_LABEL,
/*Price Cluster Attribute Block Threshold Set Attr IDs (D.4.2.2.2)*/
    E_CLD_P_ATTR_BLOCK1_THRESHOLD = 0x0100,
    E_CLD_P_ATTR_BLOCK2_THRESHOLD,
    ...
    ...
    E_CLD_P_ATTR_BLOCK15_THRESHOLD,
/*Price Cluster Attribute Block Period Set Attr IDs (D.4.2.2.3)*/
    E_CLD_P_ATTR_START_OF_BLOCK_PERIOD = 0x0200,
    E_CLD_P_ATTR_BLOCK_PERIOD_DURATION,
    E_CLD_P_ATTR_THRESHOLD_MULTIPLIER,
    E_CLD_P_ATTR_THRESHOLD_DIVISOR,
/*Price Cluster Attribute Commodity Set Attr IDs (D.4.2.2.4)*/
    E_CLD_P_ATTR_COMMODITY_TYPE = 0x0300,
    E_CLD_P_ATTR_STANDING_CHARGE,
    E_CLD_P_ATTR_CONVERSION_FACTOR,
    E_CLD_P_ATTR_CONVERSION_FACTOR_TRAILING_DIGIT,
    E_CLD_P_ATTR_CALORIFIC_VALUE,
    E_CLD_P_ATTR_CALORIFIC_VALUE_UNIT,
    E_CLD_P_ATTR_CALORIFIC_VALUE_TRAILING_DIGIT,
/* Price Cluster Attribute Block Price Information Set Attr IDs (D.4.2.2.5)*/
    E_CLD_P_ATTR_NOTIER_BLOCK1_PRICE = 0x0400,
    E_CLD_P_ATTR_NOTIER_BLOCK2_PRICE,
    ...
    ...
```

(continues on next page)

(continued from previous page)

```
    E_CLD_P_ATTR_NOTIER_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER1_BLOCK1_PRICE = 0x0410,
    ...
    E_CLD_P_ATTR_TIER1_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER2_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER2_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER3_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER3_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER4_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER4_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER5_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER5_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER6_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER6_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER7_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER7_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER8_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER8_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER9_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER9_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER10_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER10_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER11_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER11_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER12_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER12_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER13_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER13_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER14_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER14_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER15_BLOCK1_PRICE,
    ...
    E_CLD_P_ATTR_TIER15_BLOCK16_PRICE
/* Price Cluster Billing Period Information Set Attr IDs */
    E_CLD_P_ATTR_START_OF_BILLING_PERIOD = 0x700,
    E_CLD_P_ATTR_BILLING_PERIOD_DURATION
} teCLD_SM_PriceAttributeID;
```

Parent topic: Enumerations

'Price Event' Enumerations The event types generated by the Price cluster are enumerated in the teSE_PriceCallBackEventType structure below:

```
typedef enum PACK
{
    E_SE_PRICE_TABLE_ADD =0x00,
```

(continues on next page)

(continued from previous page)

```
    E_SE_PRICE_TABLE_ACTIVE,
    E_SE_PRICE_GET_CURRENT_PRICE_RECEIVED,
    E_SE_PRICE_TIME_UPDATE,
    E_SE_PRICE_ACK_RECEIVED,
    E_SE_PRICE_NO_PRICE_TABLES,
    E_SE_PRICE_READ_BLOCK_PRICING,
    E_SE_PRICE_BLOCK_PERIOD_TABLE_ACTIVE,
    E_SE_PRICE_NO_BLOCK_PERIOD_TABLES,
    E_SE_PRICE_BLOCK_PERIOD_ADD,
    E_SE_PRICE_READ_BLOCK_THRESHOLDS,
    E_SE_PRICE_CONVERSION_FACTOR_TABLE_ACTIVE,
    E_SE_PRICE_CONVERSION_FACTOR_ADD,
    E_SE_PRICE_CALORIFIC_VALUE_TABLE_ACTIVE,
    E_SE_PRICE_CALORIFIC_VALUE_ADD,
    E_SE_PRICE_CBET_ENUM_END
} teSE_PriceCallBackEventType;
```

The above event types are described in *Table 56* below.

Note: For further details on Price events, refer to Section 40.8.

Event Type Enumeration	Description
E_SE_PRICE_TABLE_ADD	Generated when a new scheduled price is added to the local price list
E_SE_PRICE_TABLE_ACTIVE	Generated when a new price becomes active or the active price expires
E_SE_PRICE_GET_CURRENT_I	Generated on the server when a Get Current Price command is received from a client
E_SE_PRICE_TIME_UPDATE	Generated on a client when a Publish Price command is received from the server
E_SE_PRICE_ACK_RECEIVED	Generated on a server when a Price Acknowledgment command is received from a client
E_SE_PRICE_NO_PRICE_TABL	Generated when an active price expires, is deleted from the price list and the list becomes empty
E_SE_PRICE_READ_BLOCK_PI	Reserved for future use (for Block mode)
E_SE_PRICE_BLOCK_PERIOD_	Reserved for future use (for Block mode)
E_SE_PRICE_NO_BLOCK_PERI	Reserved for future use (for Block mode)
E_SE_PRICE_BLOCK_PERIOD_	Reserved for future use (for Block mode)
E_SE_PRICE_READ_BLOCK_TI	Reserved for future use (for Block mode)
E_SE_PRICE_CONVERSION_FA	Generated when a new conversion factor value becomes active
E_SE_PRICE_CONVERSION_FA	Generated when a new conversion factor entry is advertised by the ESP via a Publish Conversion Factor command
E_SE_PRICE_CALORIFIC_VALI	Generated when a new calorific value becomes active
E_SE_PRICE_CALORIFIC_VALI	Generated when a new calorific value entry is advertised via a Publish Calorific Value command

Parent topic: Enumerations

'Calorific Value Unit' Enumerations The possible units for the calorific value attribute of the Price cluster are enumerated in the tsSE_PriceCalorificValueUnits structure below:

```
typedef enum PACK
{
    E_SE_MEGA_JOULES_METER_CUBE  = 0x01,
    E_SE_MEGA_JOULES_KILOGRAM = 0x02
```

(continues on next page)

(continued from previous page)

```
} tsSE_PriceCalorificValueUnits;
```

The above enumerations are described in *Table 57* below.

Enumeration	Description
E_SE_MEGA_JOULES_METER_CUBE	Calorific value measured in MJ/m3
E_SE_MEGA_JOULES_KILOGRAM	Calorific value measured in MJ/kg

Parent topic: Enumerations

Parent topic:*Price Cluster*

Compile-time options This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the Price cluster.

The Price cluster is enabled by defining CLD_PRICE.

Client and server versions of the cluster are defined by PRICE_CLIENT and PRICE_SERVER, respectively.

Price List Size The maximum number of prices that can be stored in the price list on a server and client defaults to five and two respectively. These default values can be over-ridden by assigning values to the corresponding macro below:

- SE_PRICE_NUMBER_OF_SERVER_PRICE_RECORD_ENTRIES
- SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES

Price Tier Label Attribute Set The maximum number of supported Price Tier Label Attribute Sets can be defined by assigning a value between 1 and 15 (inclusive) to CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT.

Block Threshold Attribute Set The maximum number of supported Block Threshold Attribute Sets can be defined by assigning a value between 1 to 15 (inclusive) to CLD_P_ATTR_BLOCK_THRESHOLD_MAX_COUNT.

Block Price Information Attribute Set The maximum number of supported Block Price Information Attribute Sets can be defined by assigning a value (the maximum of which is shown below in brackets) to each of the following:

- CLD_P_ATTR_NO_TIER_BLOCK_PRICES_MAX_COUNT (16)
- CLD_P_ATTR_NUM_OF_TIERS_PRICE (15)
- CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE (16)

Conversion Factor (Gas Only) Conversion factor in the Price cluster is enabled by defining the macro PRICE_CONVERSION_FACTOR.

The attributes for conversion factor are enabled by defining the following macros:

- CLD_P_ATTR_CONVERSION_FACTOR
- CLD_P_ATTR_CONVERSION_FACTOR_TRAILING_DIGIT

The default value of the maximum number of entries that can be stored in the conversion factor list which is maintained on the Price cluster server and client is 2. This value can be over-ridden by assigning another value to the macro:

SE_PRICE_NUMBER_OF_CONVERSION_FACTOR_ENTRIES

Calorific Value (Gas Only) Calorific value in the Price cluster is enabled by defining the macro PRICE_CALORIFIC_VALUE.

The attributes for calorific value are enabled by defining the following macros:

- CLD_P_ATTR_CALORIFIC_VALUE
- CLD_P_ATTR_CALORIFIC_VALUE_UNIT
- CLD_P_ATTR_CALORIFIC_VALUE_TRAILING_DIGIT

The default value of the maximum number of entries that can be stored in the calorific value list which is maintained on the server and client is 2. This value can be over-ridden by assigning another value to the macro:

SE_PRICE_NUMBER_OF_CALORIFIC_VALUE_ENTRIES

Parent topic:*Price Cluster*

Demand-Response and Load Control Cluster This chapter outlines the Demand-Response and Load Control (DRLC) cluster. The cluster is able to receive load control requests from the utility company and act upon them by controlling an attached appliance, such as a heater or pump - this is the 'demand-response' functionality.

The DRLC cluster has a Cluster ID of 0x0701.

Overview The DRLC cluster is required in ZigBee devices as indicated in the table below.

	Server-side	Client-side
Mandatory in...	ESP Load Control Device	PCT
Optional in...	Smart Appliance	IPD

The ESP acts as the DRLC cluster server, since it is the device which receives Load Control Events (LCEs) from the utility company via the backhaul network. Other devices act as clients and receive the LCEs forwarded by the ESP:

- An IPD would normally display a list of LCEs to allow the consumer to manually modify consumption.
- A Load Control Device, PCT or Smart Appliance would participate in an LCE by automatically adjusting the consumption of the device.

Devices that participate in an LCE must report their participation back to the ESP. Participation may result in the consumer receiving a credit on their utility bill.

Note: In the current NXP implementation, the DRLC cluster client is contained within an IPD only. This illustrates how to incorporate the DRLC cluster in other devices which need to participate in LCEs.

The LCEs contain a time-stamp. Therefore, devices which support the DRLC cluster client and which participate in LCEs must implement the Time cluster and maintain a real-time clock.

The DRLC cluster is enabled by defining CLD_DRLC in the **zcl_options.h** file. Further compile-time options for the DRLC cluster are detailed in Section 41.12.

Parent topic:*Demand-Response and Load Control Cluster*

DRLC Cluster structure and attributes The DRLC cluster has no server attributes but has client attributes that are contained in the following tsCLD_DRLC structure:

```
typedef struct
{
    uint8       u8UtilityEnrolmentGroup;
    uint8       u8StartRandomizeMinutes;
    uint8       u8StopRandomizeMinutes;
    uint16      u16DeviceClassValue;
} tsCLD_DRLC;
```

where:

- u8UtilityEnrolmentGroup identifies the 'enrolment' group to which the device belongs, where a group of devices is defined by the utility company in order to aid load management in a large system. The default value of 0x00 is used to indicate membership of all groups.
- u8StartRandomizeMinutes specifies the largest random delay, in minutes, that can be applied to the start of a Load Control Event (so a random delay, no greater than this value, will be applied to an individual event). The valid range of values is 0x00 to 0x3C (0 to 60 mins), where 0x00 indicates that no delay is to be applied.
- u8StopRandomizeMinutes specifies the largest random delay, in minutes, that can be applied to the end of a Load Control Event (so a random delay, no greater than this value, will be applied to an individual event). The valid range of values is 0x00 to 0x3C, where 0x00 indicates that no delay is to be applied.
- u16DeviceClassValue is a bitmap specifying the relevant device classes (for example, water heater and pool pump). Enumerations are provided for the device classes and are detailed in Section 41.10.1. If more than one device class is required, the relevant enumerations can be bitwise-ORed.

Note: It may be desirable to refuse write access to the u16DeviceClassValue attribute on a device. To do this, when a 'write attributes' request is received and an E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE event is generated for this attribute, the application should set the eAttributeStatus field of the event to E_ZCL_DENY_ATTRIBUTE_ACCESS.

Parent topic:*Demand-Response and Load Control Cluster*

Initialization Provided that the DRLC cluster is enabled in the compile-time options (see Section41.12), the cluster is automatically initialized when the ZCL is initialized and the ZigBee device is registered in the application - that is, by calling **eZCL_Initialise()** and the relevant endpoint registration function for the device, for example:

- **eSE_RegisterEspEndPoint()** on a standalone ESP (cluster server)
- **eSE_RegisterIPDEndPoint()** on an IPD (cluster client)

As part of this initialization, the DRLC cluster is created and, on the ESP, a DRLC timer server is registered to support time-stamps in the LCEs.

A DRLC cluster client must also perform a number of other initialization steps in order to establish communication with the cluster server. These are described below.

1. **Set 'device class' attribute:** The value of the 'device class' attribute (see Section 41.2) must be set immediately after **eSE_RegisterIPDEndPoint()** is called and before the network is started.
2. **Bind to server:** A non-sleeping client should bind its endpoint to the server using the ZigBee PRO API function **ZPS_eAplZdpBindUnbindRequest()**. This allows the server to send out unsolicited LCEs to the client.

- Before this binding can take place, the client must obtain the IEEE/MAC address of the ESP/server. This can be achieved by first using the function **ZPS_eAplZdpMatchDescRequest()** to find the ESP/Server and to obtain its network address. The function **ZPS_eAplZdpIeeeAddrRequest()** can then be used to obtain the corresponding IEEE/MAC address. Once found, both addresses must be added to the local Address Map using the function **ZPS_eAplZdoAddAddrMapEntry()**.
- All four of the above ZPS functions are described in the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

3. **Synchronize time with ESP:** A client should synchronize ZCL time with the ESP using the Time cluster as soon as initialization is complete. It is not possible to process unsolicited LCEs with a 'start-time of now' until ZCL time has been synchronized.

Once the clients have been set up, the ESP/server may need to configure the enrolment groups and randomization attributes of the DRLC clients (see Section41.2). The ESP may use one of the following mechanisms to determine when a DRLC client has come on-line:

- A Get Scheduled Events message is received from a new client
- A Report Event Status message is received from a new client
- A binding request is received from a new client

Parent topic:*Demand-Response and Load Control Cluster*

Load Control Events (LCEs) The Load Control Event (LCE) is an instruction, which originates from the utility company, to schedule a temporary adjustment of consumption in devices that support the DRLC cluster. The contents of an LCE are outlined in Section 41.4.1.

An LCE is sent from the utility company to the DRLC server (ESP) of a ZigBee network, from where it is passed to DRLC clients. The LCEs are held in lists on the server and clients, as described in Section 41.4.2.

LCE handling is described in Section 41.5.

LCE Contents The information contained in an LCE includes:

- LCE ID (provided by utility company)
- Target device class and enrolment group
- Start-time
- Duration
- Criticality level
- Required adjustment(s)
- Randomization requirements (for start-time and end-time)

For a full list and description of the LCE data, refer to the description of the LCE structure `tsSE_DRLCLoadControlEvent` in Section 41.11.1.

Parent topic: Load Control Events (LCEs)

LCE Lists The DRLC cluster server and clients each hold the following lists of LCEs:

- **Active list:** Contains LCEs that are currently being executed - it is possible for more than one LCE to be active at the same time, provided that their device classes and enrolment groups do not clash.
- **Scheduled list:** Contains LCEs that are due to be executed in the future - that is, their start-time is later than the current time.
- **Cancelled list:** Contains LCEs that have been canceled with a randomized end-time and whose random end-time has not yet been reached.
- **Deallocated list:** Contains expired LCEs and therefore a record of the free storage for LCEs - used internally by the cluster (and not by the application).

A new LCE is first added to the Scheduled list, unless it has a 'start-time of now' in which case it is added to the Active list. An LCE in the Scheduled list is automatically moved to the Active list at the scheduled start-time (or at the randomized start-time). At the end of an active LCE, it is automatically moved to the Deallocated list. However, an active LCE which is canceled with a randomised end-time is automatically moved to the Cancelled list, where it stays until the end-time has been reached (when it is moved to the Deallocated list).

The addition of a new LCE on the cluster server is performed by the server application, as described in Section 41.5.1, but is done automatically by the cluster on the clients. All other operations on LCE lists, apart from cancelation (see Section 41.5.3), are performed automatically by the cluster on both server and client.

Functions are provided to access entries in the local LCE lists:

- **eSE_DRLCGetLoadControlEvent()** can be used to obtain a particular LCE entry (with specified list index) in any one of the local lists
- **eSE_DRLCFindLoadControlEvent()** can be used to search for and obtain a particular LCE (with specified ID) in any of the local lists

Parent topic: Load Control Events (LCEs)

Parent topic:*Demand-Response and Load Control Cluster*

LCE Handling LCEs are handled on the DRLC cluster server and clients as described in Section41.5.1 and Section 41.5.2 respectively. Canceling LCEs is described in Section41.5.3.

Note: The DRLC callback events referred to in this section are further described in Section 41.7 and are handled by the callback function that is registered as part of the device endpoint registration.

LCE Handling on Server When a new LCE is received from the utility company, it is the responsibility of the application on the ESP (DRLC cluster server) to add this LCE to the local 'Scheduled' list (or to the 'Active' list, if the LCE has a 'start-time of now'). This addition is performed using the function **eSE_DRLCAddLoadControlEvent()**, which also sends the LCE (unsolicited) to the cluster clients. The LCE should normally be sent to all client endpoints with which the cluster server has been bound (see Section 41.3).

Note:

1. **Note 1:**Following the initial reception of LCEs from the utility company, the addition of these LCEs to the list(s) through **eSE_DRLCAddLoadControlEvent()**can be done after calling **eSE_RegisterEspMeterEndPoint()** or **eSE_RegisterEspEndPoint()** but before calling **ZPS_eAplZdoStartStack()**.
2. **Note 2:** On receiving an LCE, the client checks the device class and enrollment group specified within the LCE, and only accepts the LCE if these values match the corresponding DRLC cluster attributes held locally (see Section 41.2).

The cluster server also automatically responds to Get Scheduled Events messages from cluster clients that need current and future LCEs (see Section 41.5.2.2).

Parent topic:LCE Handling

LCE Handling on Clients The sub-sections below describe the various LCE handling activities that take place on a DRLC cluster client.

LCE Activation and De-activation On receiving a new LCE from the DRLC cluster server, a cluster client first checks the device class and enrollment group specified within the LCE. If they do not match those of the local device (see DRLC attributes in Section 41.2), the LCE is discarded.

Note: A DRLC cluster client can opt out of an individual LCE using the **eSE_DRLCSetEventUserOption()** function.

Generally, a valid LCE received from the cluster server is automatically added to the 'Scheduled' list on the client - the E_SE_DRLC_EVENT_COMMAND callback event containing the command SE_DRLC_LOAD_CONTROL_EVENT is generated on the client to indicate that this has been done. However, if the LCE has a 'start-time of now', it is added directly to the 'Active' list, provided that the start-time is not randomized (see below).

If a new LCE is successfully added to the Scheduled (or Active) list, the client sends a Report Event Status message to the server to confirm acceptance of the LCE.

When the start-time of an LCE in the 'Scheduled' list is reached, the LCE is automatically moved to the 'Active' list. The E_SE_DRLC_EVENT_ACTIVE callback event is generated on the client to indicate that this has been done, allowing the application to make the required load adjustment. However, if a randomized start-time is enabled (in the LCE), the move to the 'Active' list is delayed by a random time interval that is no greater than the maximum defined by the cluster attribute u8StartRandomizeMinutes (see Section 41.2).

When the duration of the active LCE has expired, the LCE is automatically moved to the 'De-allocated' list - the E_SE_DRLC_EVENT_EXPIRED callback event is generated on the client to indicate that this has been done, allowing the application to restore the load to the previous level. However, if a randomized end-time is enabled (in the LCE), the move to the 'Deallocated' list is delayed by a random time interval that is no greater than the maximum defined by the cluster attribute u8StopRandomizeMinutes (see Section 41.2).

Note: The above randomize attributes of the DRLC cluster also allow LCE start-time and end-time randomization to be disabled for all LCEs on the local device. If this is the case, randomization settings within the LCE itself are ignored.

Parent topic:LCE Handling on Clients

Getting Scheduled Events The application on the DRLC cluster client can send a Get Scheduled Events message to the cluster server in order to obtain relevant current and future LCEs. This message may be used in the following situations:

- On a non-sleeping device, the application may send this message:
  - immediately after binding with the cluster server in order to get the initial LCEs (subsequent LCEs are received unsolicited from the server).
  - at other times in order to top up its LCE list, if it has previously discarded an LCE due to lack of storage.
- On a sleeping device (End Device), the application may send this message on waking from sleep in order to obtain new LCEs that were distributed by the cluster server during sleep (and therefore not received).

The Get Scheduled Events message can be sent from a client using the function eSE_DRLCGetScheduledEventsSend(). The message includes the earliest start-time of the LCEs of interest, where zero is used to indicate all LCEs - for a sleeping End Device, this time should be set to zero or the current time, in case there are replacements on the server for LCEs already in the client's lists. The message also allows the maximum number of returned LCEs to be specified, where zero is used to indicate all LCEs.

Note: The arrival of the Get Scheduled Events message results in the generation of the E_SE_DRLC_EVENT_COMMAND callback event, containing a DRLC_GET_SCHEDULED_EVENTS command on the cluster server. However, the cluster responds to the message automatically.

On receiving the requested LCEs from the cluster server, the cluster client automatically updates the local LCE lists with the reported LCEs.

Parent topic:LCE Handling on Clients

Reporting LCE Actions to Server By default, a DRLC cluster client sends a Report Event Status message to the cluster server when an LCE is actioned on the client - that is, when an LCE is moved between lists on the client, such as from 'Scheduled' to 'Active' or from 'Active' to 'Deallocated' (see Section 41.4.2). Details of the actioned LCE are sent in a tsSE_DRLCReportEvent structure (see Section 41.11.4). The nature of the action is indicated in this structure using an enumeration (see Section 41.10.8).

Note: The DRLC cluster server is informed of the arrival of a Report Event Status message via the callback event **E_SE_DRLC_EVENT_COMMAND**, containing a SE_DRLC_REPORT_EVENT_STATUS command. The ESP/server may inform the utility company of the reported status - if the message cannot be forwarded immediately then it must be buffered by the application.

If a DRLC cluster client opts out of a particular LCE using the function **eSE_DRLCSetEventUserOption()**, a Report Event Status message is sent to the cluster server to indicate this. On reaching the end-time of the LCE, another Report Event Status message is sent to the server to confirm that the LCE has completed without the participation of the local client.

Parent topic:LCE Handling on Clients

Over-riding LCE Settings The client application can over-ride certain aspects of an LCE using the function **eSE_DRLCSetEventUserData()**, which allows load control data values to be modified, including:

- Criticality level
- Cooling temperature set-point
- Heating temperature set-point
- Load adjustment percentage
- Duty cycle

For example, the ESP/server may request an HVAC device to set its cooling level to 24oC, but the user may choose to over-ride this with a cooling level of 20oC. The above data values and their formats are detailed in the LCE structure description in Section 41.11.1.

The function **eSE_DRLCSetEventUserData()** modifies one load control data value on each call. Therefore, in order to modify more than one data value, the function must be called multiple times.

When a change is made, the cluster client automatically notifies the cluster server by sending a Report Event Status message containing the change.

Parent topic:LCE Handling on Clients

Parent topic:LCE Handling

Canceling LCEs An LCE can be canceled, in which case it is moved to the 'Deallocated' list (possibly via the 'Cancelled' list - see below). A cancellation can only be performed from the DRLC cluster server and is normally sent to all client endpoints that have been bound to the server. Two functions are provided which can be called on the cluster server to perform LCE cancelations:

- **eSE_DRLCCancelLoadControlEvent()** is used to cancel a particular LCE
- **eSE_DRLCCancelAllLoadControlEvents()** is used to cancel all LCEs

Cancellation involves removing the LCE(s) from the 'Scheduled' or 'Active' lists on the cluster server and clients, which is done automatically by the cluster. As a result, the callback event E_SE_DRLC_EVENT_COMMAND is generated, containing a LOAD_CONTROL_EVENT_CANCEL or LOAD_CONTROL_EVENT_CANCEL_ALL command, as appropriate. This indicates whether the cancellation with immediate effect or a random delay is applied:

- If the cancellation is with immediate effect, the application should stop load control for the relevant device(s).
- If a random delay is to be applied to the cancellation, the cluster puts the LCE in the 'Cancelled' list until the delay has expired, when the LCE is moved to the 'Deallocated' list. Another E_SE_DRLC_EVENT_COMMAND callback event containing the command LOAD_CONTROL_EVENT_CANCEL or LOAD_CONTROL_EVENT_CANCEL_ALL is then generated, this time indicating immediate cancellation. The application should now stop load control for the relevant device(s).

Parent topic:LCE Handling

Parent topic:*Demand-Response and Load Control Cluster*

Message Signing (Security) As a security measure, Report Event Status messages can be signed by the DRLC cluster client for non-repudiation purposes (to provide the utility company with evidence that the cluster client sent the message). On the DRLC cluster client, the process involves generating a hash value which is based on the content of the message, then using this value in combination with a device's private key to generate a signature, which is then appended to the message to be sent to the ESP.

Upon message reception on the ESP, the hash value is recalculated based on the received message and then used in conjunction with the public key of the message originator (derived from the originator's certificate) to check the appended signature. To facilitate this checking, the ESP must store the certificates of any nodes that send Report Event Status messages which require verification.

Note:

1. It is recommended that signatures are supported by applications for backward compatibility.
2. Signature fields are included in the Report Event Status structure, detailed in Section 41.11.4.

Message signing must be enabled at compile-time, as described in Section 41.12.

Parent topic:*Demand-Response and Load Control Cluster*

DRLC Events The DRLC cluster has its own events that are handled through the callback mechanism described in *Chapter 3*. If a device uses the DRLC cluster then DRLC event handling must be included in the callback function for the associated endpoint - for example:

- For an ESP (cluster server), this callback function is registered through **eSE_RegisterEspMeterEndPoint()** or **eSE_RegisterEspEndPoint()**
- For an IPD (cluster client), this callback function is registered through **eSE_RegisterIPDEndPoint()**

The relevant callback function is then invoked when a DRLC event occurs.

For a DRLC event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to a tsSE_DRLCCallBackMessage structure which contains the DRLC parameters:

```
typedef struct
{
 teSE_DRLCCallBackEventType  eEventType;
  uint8   u8CommandId;
  teSE_DRLCStatus  eDRLCStatus;
  uint32  u32CurrentTime;
  union {
  tsSE_DRLCLoadControlEvent      sLoadControlEvent;
  tsSE_DRLCCancelLoadControlEvent  sCancelLoadControlEvent;
  tsSE_DRLCCancelLoadControlAllEvent  sCancelLoadControlAllEvent;
  tsSE_DRLCReportEvent        sReportEvent;
  tsSE_DRLCGetScheduledEvents   sGetScheduledEvents;
   } uMessage;
} tsSE_DRLCCallBackMessage;
```

Information on the elements of the above structure is provided in the sub-sections below.

Event and Command Types The eEventType field of the tsSE_DRLCCallBackMessage structure above specifies the type of DRLC event that has been generated - these event types are enumerated in the teSE_DRLCCallBackEventType structure, described below.

Note: The u8CommandId field of the tsSE_DRLCCallBackMessage structure is only required for a DRLC event of type E_SE_DRLC_EVENT_COMMAND (see below).

```
typedef enum PACK
{
   E_SE_DRLC_EVENT_API =0x00,
   E_SE_DRLC_EVENT_COMMAND,
   E_SE_DRLC_EVENT_ACTIVE,
   E_SE_DRLC_EVENT_EXPIRED,
   E_SE_DRLC_EVENT_CANCELLED,
   E_SE_DRLC_EVENT_ENUM_END,
} teSE_DRLCCallBackEventType;
```

E_SE_DRLC_EVENT_API The E_SE_DRLC_EVENT_API event is reserved for internal use.

E_SE_DRLC_EVENT_COMMAND The E_SE_DRLC_EVENT_COMMAND event is generated when a command has been received on either the server or client. In the tsSE_DRLCCallBackMessage structure, the u8CommandId field is used to indicate the corresponding command - one of:

Command	Description
SE_DRLC_LOAD_C(	Generated on a client when a new LCE has been received from the server and added to the 'Scheduled' (or 'Active') list - the LCE is included in the

uMessage.LoadControlEvent field of the tsSE_DRLCCallBackMessage structure

| |SE_DRLC_LOAD_CONTROL_EVENT_CANCEL *|Generated on a client when a command has been received to cancel an LCE and the LCE has been moved to the 'Cancelled' or ' Deallocated' list - which list depends on whether an immediate or randomized end-time is specified in the

uMessage.sCancelLoadControlEvent field of the tsSE_DRLCCallBackMessage structure

| |SE_DRLC_LOAD_CONTROL_EVENT_CANCEL_ALL *|Generated on a client when a command has been received to cancel all LCEs and the LCEs have been moved to the 'Cancelled' or ' Deallocated' list - which list depends on whether an immediate or randomized end-time is specified in the

uMessage.sCancelLoadControlAllEvent field of the tsSE_DRLCCallBackMessage structure

| |SE_DRLC_REPORT_EVENT_STATUS **|Generated on the server when a Report Event Status message is received from a client - the contents of the report are included in the uMessage.sReportEvent field of the tsSE_DRLCCallBackMessage structure| |SE_DRLC_GET_SCHEDULED_EVENTS **|Generated on the server when a Get Scheduled Events message is received from a client - the contents of the request are included in the

uMessage.sGetScheduledEvents field of the tsSE_DRLCCallBackMessage structure

|

* If an LCE cancellation with a randomized end-time is required, the LCE is first moved to the 'Cancelled' list and the event is generated with randomized end-time specified. When the randomized end-time has been reached, the LCE is moved to the 'Deallocated' list and the event is generated again but with an immediate end-time specified. The application must then stop the corresponding load control.

** The server can identify which client has sent a Report Event Status or Get Scheduled Events message by examining the pZPSevent field of the tsZCL_CallBackEvent structure that contains the message.

E_SE_DRLC_EVENT_ACTIVE The E_SE_DRLC_EVENT_ACTIVE event is generated when an LCE has been moved from the 'Scheduled' list to the 'Active' list (see Section 41.4.2). The activated LCE is included in the uMessage.LoadControlEvent field of the tsSE_DRLCCallBackMessage structure.

E_SE_DRLC_EVENT_EXPIRED The E_SE_DRLC_EVENT_EXPIRED event is generated when an LCE has been moved from the 'Active' list (see Section 41.4.2). The expired LCE is included in the uMessage.LoadControlEvent field of the tsSE_DRLCCallBackMessage structure.

E_SE_DRLC_EVENT_CANCELLED The E_SE_DRLC_EVENT_CANCELLED event is generated when an LCE has been put in the 'Cancelled' list (see Section 41.4.2) as the result of an LCE 'cancel' or 'cancel all' command. Information on the cancelled LCE(s) is included in the uMessage.sCancelLoadControlEvent or uMessage.sCancelLoadControlAllEvent field of the tsSE_DRLCCallBackMessage structure, as appropriate.

Parent topic:DRLC Events

Other Elements of tsSE_DRLCCallBackMessage In addition to the fields eEventType and u8CommandId described in Section 41.7.1, the tsSE_DRLCCallBackMessage structure contains the following elements.

eDRLCStatus The eDRLCStatus field indicates the status returned from the command that has been executed (the command identified in u8CommandId). The status codes are enumerated in the teSE_DRLCStatus structure, shown below and described in Section 41.9.

```
typedef enum PACK
{
  E_SE_DRLC_DUPLICATE_EXISTS  = 0x80,
  E_SE_DRLC_EVENT_LATE,
```

(continues on next page)

(continued from previous page)

```
  E_SE_DRLC_EVENT_NOT_YET_ACTIVE,
  E_SE_DRLC_EVENT_OLD,
  E_SE_DRLC_NOT_FOUND,
  E_SE_DRLC_EVENT_NOT_FOUND,
  E_SE_DRLC_EVENT_IGNORED,
  E_SE_DRLC_CANCEL_DEFERRED,
  E_SE_DRLC_BAD_DEVICE_CLASS,
  E_SE_DRLC_BAD_CRITICALITY_LEVEL,
  E_SE_DRLC_DURATION_TOO_LONG,
  E_SE_DRLC_ENUM_END
} teSE_DRLCStatus;
```

u32CurrentTime The u32CurrentTime field contains the time (UTC) at which the event was generated.

uMessage This field is a union of structures, containing a structure for each of the DRLC command payloads. The valid structure in the event is defined by the value of u8CommandId (refer to the description of the E_SE_DRLC_EVENT_COMMAND event in Section 41.7.1).

Parent topic:DRLC Events

Parent topic:*Demand-Response and Load Control Cluster*

Functions The following DRLC cluster functions are provided:

- eSE_DRLCCreate
- eSE_DRLCAddLoadControlEvent
- eSE_DRLCGetScheduledEventsSend
- eSE_DRLCCancelLoadControlEvent
- eSE_DRLCCancelAllLoadControlEvents
- eSE_DRLCSetEventUserOption
- eSE_DRLCSetEventUserData
- eSE_DRLCGetLoadControlEvent
- eSE_DRLCFindLoadControlEvent

eSE_DRLCCreate

```
teZCL_Status eSE_DRLCCreate(
    bool_t bIsServer,
    uint8 u8NumberOfRecordEntries,
    uint8 *pu8AttributeControlBits,
    tsZCL_ClusterInstance *psClusterInstance,
    tsZCL_ClusterDefinition *psClusterDefinition,
    tsSE_DRLCCustomDataStructure
    *psCustomDataStructure,
    tsSE_DRLCLoadControlEventRecord
    *psDRLCLoadControlEventRecord,
    void *pvEndPointSharedStructPtr);
```

Description This function creates an instance of the DRLC cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates a DRLC cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to *Appendix D.*

Note: This function must not be called for an endpoint on which a standard ZigBee device (example, IPD) will be used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in the *ZigBee Devices User Guide (JNUG3131)*.

When used, this function must be the first DRLC cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8)** for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the DRLC cluster.

The function initializes the array elements to zero.

Parameters

- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *u8NumberOfRecordEntries*Number of LCEs that can be stored in the LCE list, one of:
  - SE_DRLC_NUMBER_OF_SERVER_LOAD_CONTROL_ENTRIES
  - SE_DRLC_NUMBER_OF_CLIENT_LOAD_CONTROL_ENTRIES
- *pu8AttributeControlBits*Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
- *psClusterInstance*Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *psClusterDefinition*Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the DRLC cluster. This parameter can refer to a pre-filled structure called sCLD_DRLC which is provided in the **DRLC.h** file.
- *psCustomDataStructure*Pointer to structure which contains custom data for the DRLC cluster. This structure is used for internal data storage. No knowledge of the fields of this structure is required.
- *psDRLCLoadControlEventRecord*
- Pointer to a structure in which an LCE is stored
- *pvEndPointSharedStructPtr*Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_DRLC which defines the attributes of DRLC cluster. The function initializes the attributes with default values.

Returns

- E_ZCL_SUCCESS

- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eSE_DRLCAddLoadControlEvent

```
teSE_DRLCStatus eSE_DRLCAddLoadControlEvent(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address psDestinationAddress,
    tsSE_DRLCLoadControlEvent *psLoadControlEvent,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on the DRLC cluster server to add an LCE (received from the utility company) to the 'Scheduled' list. The function also sends the LCE to the specified DRLC cluster client endpoints, where it will also be added to the 'Scheduled' list. Note that the LCE will be added to the 'Active' lists on the relevant devices if a 'start-time of now' is specified in the LCE.

The LCE should normally be sent to client endpoints that have been previously bound to the cluster server. This is done by specifying an address type of E_ZCL_AM_BOUND in the tsZCL_Address structure - in this case, the address field of this structure and the destination endpoint in the function call are both ignored.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the LCE is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the LCE is sent. Note that this parameter is ignored when sending to address types E_ZCL_AM_BOUND and E_ZCL_AM_GROUP
- *psDestinationAddress*: Pointer to a ZCL structure containing the address of the remote node to which the LCE is sent
- *psLoadControlEvent*: Pointer to a structure (see Section 41.11.1) which contains the LCE to be added and sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the packet sent

Returns Any relevant DRLC return code listed in Section 41.9 or ZCL return code listed in Section 7.2

Parent topic: Functions

eSE_DRLCGetScheduledEventsSend

```
teSE_DRLCStatus eSE_DRLCGetScheduledEventsSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address psDestinationAddress,
    tsSE_DRLCGetScheduledEvents *psGetScheduledEvents,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on a DRLC cluster client to send a Get Scheduled Events message to the cluster server in order to request a list of scheduled (and active) LCEs. The function can be used to obtain the initial schedule of LCEs and to update the local LCE lists during operation (for example, if an End Device has been sleeping and has missed unsolicited LCE updates) - refer to Section 41.5.2.2 for more information on the use of this function.

As part of this function call, a tsSE_DRLCGetScheduledEvents structure must be provided which specifies the earliest start-time of the LCEs of interest and the maximum number of LCEs to be reported.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent (this must be the DRLC cluster server endpoint)
- *psDestinationAddress*: Pointer to a ZCL structure containing the address of the remote node to which the request is sent (this must be the address of the ESP)
- *psGetScheduledEvents*: Pointer to a structure which contains the LCE requirements of the request (see Section 41.11.2)
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns Any relevant DRLC return code listed in Section 41.9 or ZCL return code listed in Section 7.2

Parent topic: Functions

eSE_DRLCCancelLoadControlEvent

```
teSE_DRLCStatus eSE_DRLCCancelLoadControlEvent(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address psDestinationAddress,
    tsSE_DRLCCancelLoadControlEvent
    *psCancelLoadControlEvent,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on the DRLC cluster server to cancel an LCE. The LCE is cancelled locally and the cancellation is also sent to the specified DRLC cluster client endpoints. The LCE is ultimately moved to the 'Deallocated' list.

The cancellation request should normally be sent to client endpoints that have been previously bound to the cluster server. This is done by specifying an address type of E_ZCL_AM_BOUND in the tsZCL_Address structure - in this case, the address field of this structure and the destination endpoint in the function call are both ignored.

The LCE cancellation requirements are specified in the structure tsSE_DRLCCancelLoadControlEvent, including the applicable device class(es) and enrolment group(s), as well as an immediate or randomized end (for a full description of the end-time options, refer to Section 41.5.3).

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent. Note that this parameter is ignored when sending to address types E_ZCL_AM_BOUND and E_ZCL_AM_GROUP
- *psDestinationAddress*: Pointer to a ZCL structure containing the address of the remote node to which the request is sent
- *psCancelLoadControlEvent*: Pointer to a structure which contains the LCE cancellation requirements (see Section 41.11.3)
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns Any relevant DRLC return code listed in Section 41.9 or ZCL return code listed in Section 7.2

Parent topic: Functions

eSE_DRLCCancelAllLoadControlEvents

```
teSE_DRLCStatus eSE_DRLCCancelAllLoadControlEvents(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address psDestinationAddress,
    teSE_DRLCCancelControl eCancelEventControl,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on the DRLC cluster server to cancel all LCEs. The LCEs are cancelled locally and the cancellation is also sent to the specified DRLC cluster client endpoints. The LCEs are ultimately moved to the 'Deallocated' list.

The cancellation request should normally be sent to client endpoints that have been previously bound to the cluster server. This is done by specifying an address type of E_ZCL_AM_BOUND in the tsZCL_Address structure - in this case, the address field of this structure and the destination endpoint in the function call are both ignored.

The LCE cancellation end-time requirement must be specified as an immediate or randomized end (for a full description of the end-time options, refer to Section41.5.3).

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent. Note that this parameter is ignored when sending to address types E_ZCL_AM_BOUND and E_ZCL_AM_GROUP
- *psDestinationAddress*: Pointer to a ZCL structure containing the address of the remote node to which the request is sent
- *eCancelEventControl*: Enumeration indicating an immediate or randomized end, one of:
- E_SE_DRLC_CANCEL_CONTROL_IMMEDIATE
- E_SE_DRLC_CANCEL_CONTROL_USE_RANDOMISATION
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns Any relevant DRLC return code listed in Section 41.9 or ZCL return code listed in Section 7.2

Parent topic: Functions

eSE_DRLCSetEventUserOption

```
teSE_DRLCStatus eSE_DRLCSetEventUserOption(
    uint32 u32IssuerId,
    uint8 u8SourceEndPointId,
    teSE_DRLCUserEventOption eEventOption);
```

Description This function can be used on a DRLC cluster client to choose to participate or not participate in an individual LCE. By default, a client participates in an LCE, so normally this function only needs to be called if the client is to opt out of the LCE.

The function could be called following a button-press which results from a user decision to opt out of the LCE (for which information is displayed on the IPD screen).

When this function is called, a Report Event Status message is sent to the cluster server in order to indicate that the local client has opted out of the LCE. Once the LCE end-time has been reached, another Report Event Status message is sent to the server in order to confirm that the LCE has completed without the participation of the local client.

Parameters

- *u32IssuerId*: Identifier of the LCE (as issued by the utility company)
- *u8SourceEndPointId*: Number of the local endpoint where the LCE is located (endpoint corresponding to the DRLC cluster)
- *eEventOption*: Required option, one of:
  - E_SE_DRLC_EVENT_USER_OPT_IN (participate)
  - E_SE_DRLC_EVENT_USER_OPT_OUT (do not participate)

Returns Any relevant DRLC return code listed in Section 41.9 or ZCL return code listed in Section 7.2

Parent topic: Functions

eSE_DRLCSetEventUserData

```
teSE_DRLCStatus eSE_DRLCSetEventUserData(
    uint32 u32IssuerId,
    uint8 u8SourceEndPointId,
    teSE_DRLCUserEventSet eUserEventSetID,
    uint16 u16EventData);
```

Description This function can be used on a DRLC cluster client to locally modify the load control data of an LCE. Any one of the following data values can be changed:

- Criticality level
- Cooling temperature set-point
- Heating temperature set-point

- Load adjustment percentage
- Duty cycle

The function can be called multiple times to modify more than one of the above values. The data values are fully described in Section 41.11.1.

Parameters

- *u32IssuerId*: Identifier of the LCE (as issued by the utility company)
- *u8SourceEndPointId*: Number of the local endpoint where the LCE is located (endpoint corresponding to the DRLC cluster)
- *eUserEventSetID*: Identifier of the load control data item to be modified, one of:
  - E_SE_DRLC_CRITICALITY_LEVEL_APPLIED
  - E_SE_DRLC_COOLING_TEMPERATURE_SET_POINT_APPLIED
  - E_SE_DRLC_HEATING_TEMPERATURE_SET_POINT_APPLIED
  - E_SE_DRLC_AVERAGE_LOAD_ADJUSTMENT_PERCENTAGE_APPLIED
  - E_SE_DRLC_DUTY_CYCLE_APPLIED
- *u16EventData*: Value to which the specified data item is to be set (for formats of data values, refer to descriptions in Section 41.11.1)

Returns Any relevant DRLC return code listed in Section 41.9 or ZCL return code listed in Section 7.2

Parent topic: Functions

eSE_DRLCGetLoadControlEvent

```
teSE_DRLCStatus eDRLCGetLoadControlEvent(
    uint8 u8SourceEndPointId,
    uint8 u8TableIndex,
    teSE_DRLCEventList eEventList,
    tsSE_DRLCLoadControlEvent **ppsLoadControlEvent);
```

Description This function can be used to obtain an LCE from a local LCE list.

The required list must be specified as one of 'Scheduled', 'Active', 'Cancelled' and 'Deallocated'. The index of the required LCE in the list must also be specified. The index of zero is used to indicate that the LCE with the oldest start-time should be retrieved. To retrieve all the LCEs in a list, repeatedly call this function with index zero until the function indicates that there are no further LCEs in the list (returns E_SE_DRLC_EVENT_NOT_FOUND).

Parameters

- *u8SourceEndPointId*: Number of the local endpoint from which the LCE is to be retrieved (endpoint corresponding to the DRLC cluster)
- *u8TableIndex*: Index of required LCE in the specified LCE list (see below)
- *eEventList*: LCE list from which the LCE is to be retrieved, one of:
  - E_SE_DRLC_EVENT_LIST_SCHEDULED
  - E_SE_DRLC_EVENT_LIST_ACTIVE

  - E_SE_DRLC_EVENT_LIST_CANCELLED
  - E_SE_DRLC_EVENT_LIST_DEALLOCATED
- *ppsLoadControlEvent*Pointer to a pointer to a tsSE_DRLCLoadControlEvent structure to receive the obtained LCE (see Section 41.11.1)

Returns Any relevant DRLC return code listed in Section 41.9 or ZCL return code listed in Section 7.2

Parent topic: Functions

eSE_DRLCFindLoadControlEvent

```
teSE_DRLCStatus eSE_DRLCFindLoadControlEvent(
uint8 u8SourceEndPointId,
uint32 u32IssuerId,
bool_t bIsServer,
tsSE_DRLCLoadControlEvent **ppsLoadControlEvent,
teSE_DRLCEventList *peEventList);
```

Description This function can be used to obtain the specified LCE from the local LCE lists.

The required LCE must be specified in terms of its identifier issued by the utility company. The function will search all the local LCE lists, identify the list (if any) in which the LCE was found and return the found LCE.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint from which the LCE is to be retrieved (endpoint corresponding to the DRLC cluster)
- *u32IssuerId*: Identifier of the LCE to be found (as issued by the utility company)
- *bIsServer*: Cluster server or client:
  - TRUE - server
  - FALSE - client
- *ppsLoadControlEvent*Pointer to a pointer to a tsSE_DRLCLoadControlEvent structure to receive the obtained LCE (see Section 41.11.1)
- *peEventList*: Pointer to variable to receive enumerated value of the list in which the LCE was found (see Section 41.10.7)

Returns Any relevant DRLC return code listed in Section 41.9 or ZCL return code listed in Section 7.2

Parent topic: Functions

Parent topic:*Demand-Response and Load Control Cluster*

Return codes In addition to some of the ZCL status enumerations (detailed in Section 7.2), the following enumerations are returned by the DRLC cluster functions (described in Section 41.8) to indicate the outcome of the function call.

```
typedef enum PACK
{
 E_SE_DRLC_DUPLICATE_EXISTS = 0x80,
 E_SE_DRLC_EVENT_LATE,
 E_SE_DRLC_EVENT_NOT_YET_ACTIVE,
 E_SE_DRLC_EVENT_OLD,
 E_SE_DRLC_NOT_FOUND,
 E_SE_DRLC_EVENT_NOT_FOUND,
 E_SE_DRLC_EVENT_IGNORED,
 E_SE_DRLC_CANCEL_DEFERRED,
 E_SE_DRLC_BAD_DEVICE_CLASS,
 E_SE_DRLC_BAD_CRITICALITY_LEVEL,
 E_SE_DRLC_DURATION_TOO_LONG,
 E_SE_DRLC_ENUM_END
} teSE_DRLCStatus;
```

The above return codes are described in the table below.

Enumeration	Description
E_SE_DRLC_DUPLICATE_EXIST	An overlapping LCE (in time) has been found
E_SE_DRLC_EVENT_LATE	Function call refers to a time period that is earlier than the current ZCL time
E_SE_DRLC_EVENT_NOT_YET_A	Not used - reserved for future use
E_SE_DRLC_EVENT_OLD	Not used - reserved for future use
E_SE_DRLC_NOT_FOUND	LCE cannot be found in lists (used in LCE cancelation or opt out)
E_SE_DRLC_EVENT_NOT_FOUN	LCE cannot be found in lists (used when searching for an LCE)
E_SE_DRLC_EVENT_IGNORED	Not used - reserved for future use
E_SE_DRLC_CANCEL_DEFERRE	Cancellation has been processed but is deferred to act in the future
E_SE_DRLC_BAD_DEVICE_CLAS	Specified device class not recognized
E_SE_DRLC_BAD_CRITICALITY_	Specified criticaility level not recognized
E_SE_DRLC_DURATION_TOO_L	Specified duration exceeds maximum of 1440 minutes (one day)

Parent topic:*Demand-Response and Load Control Cluster*

Enumerations

'Device Class' Enumerations The device classes that are used in load control are enumerated in the teSE_DRLCDeviceClassFieldBitmap structure below:

```
typedef enum
{
  E_SE_DRLC_HVAC_COMPRESSOR_OR_FURNACE_BIT = 0x00,
  E_SE_DRLC_STRIP_BASEBOARD_HEATERS_BIT,
  E_SE_DRLC_WATER_HEATER_BIT,
  E_SE_DRLC_POOL_PUMP_SPA_JACUZZI_BIT,
  E_SE_DRLC_SMART_APPLIANCES_BIT,
  E_SE_DRLC_IRRIGATION_PUMP_BIT,
  E_SE_DRLC_MANAGED_COMMERCIAL_AND_INDUSTRIAL_LOADS_BIT,
  E_SE_DRLC_SIMPLE_MISC_LOADS_BIT,
  E_SE_DRLC_EXTERIOR_LIGHTING_BIT,
  E_SE_DRLC_INTERIOR_LIGHTING_BIT,
```

(continues on next page)

(continued from previous page)

```
  E_SE_DRLC_ELECTRIC_VEHICLE_BIT,
  E_SE_DRLC_GENERATION_SYSTEMS_BIT,
  E_SE_DRLC_DEVICE_CLASS_FIRST_RESERVED_BIT
} teSE_DRLCDeviceClassFieldBitmap;
```

The device class enumerations are listed and described in the table below.

Device Class Enumeration	Description
E_SE_DRLC_HVAC_COMPRESSOR_OR_FURNACE_BIT	HVAC compressor or furnace
E_SE_DRLC_STRIP_BASEBOARD_HEATERS_BIT	Strip/baseboard heater
E_SE_DRLC_WATER_HEATER_BIT	Water heater
E_SE_DRLC_POOL_PUMP_SPA_JACUZZI_BIT	Pool/spa/jacuzzi pump
E_SE_DRLC_SMART_APPLIANCES_BIT	Smart appliance
E_SE_DRLC_IRRIGATION_PUMP_BIT	Irrigation pump
E_SE_DRLC_MANAGED_COMMERCIAL_AND_INDUSTRIAL_	Managed Commercial & Industrial (C&I)
E_SE_DRLC_SIMPLE_MISC_LOADS_BIT	Simple miscellaneous (residential on/off)
E_SE_DRLC_EXTERIOR_LIGHTING_BIT	Exterior lighting
E_SE_DRLC_INTERIOR_LIGHTING_BIT	Interior lighting
E_SE_DRLC_ELECTRIC_VEHICLE_BIT	Electric vehicle
E_SE_DRLC_GENERATION_SYSTEMS_BIT	Generation systems
E_SE_DRLC_DEVICE_CLASS_FIRST_RESERVED_BIT	Reserved

Parent topic: Enumerations

'DRLC Event' Enumerations The event types generated by the DRLC cluster are enumerated in the teSE_DRLCCallBackEventType structure below:

```
typedef enum PACK
{
  E_SE_DRLC_EVENT_API =0x00,
  E_SE_DRLC_EVENT_COMMAND,
  E_SE_DRLC_EVENT_ACTIVE,
  E_SE_DRLC_EVENT_EXPIRED,
  E_SE_DRLC_EVENT_CANCELLED,
  E_SE_DRLC_EVENT_ENUM_END,
} teSE_DRLCCallBackEventType;
```

The above event types are described in the table below.

Event Type Enumeration	Description
E_SE_DRLC_EVENT_API	Reserved for internal use
E_SE_DRLC_EVENT_COM	Generated when a command has been received from either the cluster server or a cluster client
E_SE_DRLC_EVENT_ACT	Generated when an LCE has been added to the 'Active' list
E_SE_DRLC_EVENT_EXP	Generated when an LCE has been removed from the 'Active' list
E_SE_DRLC_EVENT_CAN	Generated when an LCE has been put in the 'Cancelled' list

DRLC events are described in more detail in Section 41.7.

Parent topic: Enumerations

'Criticality Level' Enumerations The criticality levels that are available for an LCE are enumerated in the teSE_DRLCCriticalityLevels structure below:

```
typedef enum
{
    E_SE_DRLC_RESERVED_0_CRITICALITY = 0x00,
    E_SE_DRLC_GREEN_CRITICALITY,
    E_SE_DRLC_VOLUNTARY_1_CRITICALITY,
    E_SE_DRLC_VOLUNTARY_2_CRITICALITY,
    E_SE_DRLC_VOLUNTARY_3_CRITICALITY,
    E_SE_DRLC_VOLUNTARY_4_CRITICALITY,
    E_SE_DRLC_VOLUNTARY_5_CRITICALITY,
    E_SE_DRLC_EMERGENCY_CRITICALITY,
    E_SE_DRLC_PLANNED_OUTAGE_CRITICALITY,
    E_SE_DRLC_SERVICE_DISCONNECT_CRITICALITY,
    E_SE_DRLC_UTILITY_DEFINED_1_CRITICALITY,
    E_SE_DRLC_UTILITY_DEFINED_2_CRITICALITY,
    E_SE_DRLC_UTILITY_DEFINED_3_CRITICALITY,
    E_SE_DRLC_UTILITY_DEFINED_4_CRITICALITY,
    E_SE_DRLC_UTILITY_DEFINED_5_CRITICALITY,
    E_SE_DRLC_UTILITY_DEFINED_6_CRITICALITY,
    E_SE_DRLC_FIRST_RESERVED_CRITICALITY
} teSE_DRLCCriticalityLevels;
```

The above criticality levels are described in the table below.

Criticality Level Enumeration	Description
E_SE_DRLC_RESI	Reserved for future use
E_SE_DRLC_GRE	**Green:** Indicates that there will be a significant contribution from non-green sources during the LCE - participation in the LCE is voluntary
E_SE_DRLC_VOL	**Voluntary 1-6:** Represent increasing levels of load reduction as move through levels 1 to 6, as defined by the utility company - intended to be used in a sequence of LCEs to gradually reduce loads, where participation in the LCEs is voluntary
E_SE_DRLC_VOL	
E_SE_DRLC_VOL	
E_SE_DRLC_VOL	
E_SE_DRLC_VOL	
E_SE_DRLC_EME	**Emergency:** Indicates that the LCE represents an emergency situation (normally demanding the termination of all non-essential loads, as defined by the utility company) - participation in the LCE is mandatory
E_SE_DRLC_PLA	**Planned Outage:** Indicates that the LCE represents an intentional outage (normally demanding the termination of all non-essential loads, as defined by the utility company) - participation in the LCE is mandatory
E_SE_DRLC_SERV	**Service Disconnect:** Indicates that the LCE represents a service disconnection (normally demanding the termination of all non-essential loads, as defined by the utility company) - participation in the LCE is mandatory
E_SE_DRLC_UTII	**Utility-defined 1-6:** Criticality levels completely defined by the utility company - participation in the LCE is voluntary
E_SE_DRLC_UTII	
E_SE_DRLC_UTII	
E_SE_DRLC_UTII	
E_SE_DRLC_UTII	
E_SE_DRLC_UTII	
E_SE_DRLC_FIRS	Reserved for future use

Parent topic: Enumerations

'LCE Cancellation' Enumerations The cancelation options (immediate or randomized) that are available for an LCE are enumerated in the teSE_DRLCCancelControl structure below:

```
typedef enum PACK
{
   E_SE_DRLC_CANCEL_CONTROL_IMMEDIATE =0x00,
   E_SE_DRLC_CANCEL_CONTROL_USE_RANDOMISATION =0x10
} teSE_DRLCCancelControl;
```

The above options are described in the table below.

LCE Cancellation Enumeration	Description
E_SE_DRLC_C	LCE is cancelled immediately by moving it directly to the 'Deallocated' list - a randomized end-time configured in the LCE is ignored
E_SE_DRLC_C	A random delay will be applied to the LCE cancellation, if a randomised end-time was configured in the LCE - the LCE is moved to the 'Cancelled' list where it will stay (and remain valid) until the random delay has expired, when it will be moved to the 'Deallocated' list (an upper limit on the delay is defined in the cluster - see Section41.2)

Parent topic: Enumerations

'LCE Participation' Enumerations The options to participate or not participate in an LCE are enumerated in the teSE_DRLCUserEventOption structure below:

```
typedef enum PACK
{
   E_SE_DRLC_EVENT_USER_OPT_IN =0x00,
   E_SE_DRLC_EVENT_USER_OPT_OUT
} teSE_DRLCUserEventOption;
```

The above options are described in the table below.

LCE Participation Enumeration	Description
E_SE_DRLC_EVE	User has opted not to participate in the LCE. The device sends this message and does not adjust the load when the LCE becomes active.
E_SE_DRLC_EVE	User has opted to participate in the LCE. The device only sends this message following an OPT_OUT (when the user has changed their mind and decided to participate after all)

Parent topic: Enumerations

'LCE Data Modification' Enumerations The load control data items that can be locally modified in an LCE are enumerated in the teSE_DRLCUserEventSet structure below:

```
typedef enum PACK
{
   E_SE_DRLC_CRITICALITY_LEVEL_APPLIED =0x00,
   E_SE_DRLC_COOLING_TEMPERATURE_SET_POINT_APPLIED,
   E_SE_DRLC_HEATING_TEMPERATURE_SET_POINT_APPLIED,
   E_SE_DRLC_AVERAGE_LOAD_ADJUSTMENT_PERCENTAGE_APPLIED,
   E_SE_DRLC_DUTY_CYCLE_APPLIED,
   E_SE_DRLC_USER_EVENT_ENUM_END,
} teSE_DRLCUserEventSet;
```

The above options are described in the table below (the data items are fully described in Section 41.11.1).

LCE Participation Enumeration	Description
E_SE_DRLC_CRITICALITY_LEVEL_APPLIED	Specifies that 'criticality level' is to be modified.
E_SE_DRLC_COOLING_TEMPERATURE_SET_PC	Specifies that 'cooling temperature set-point' is to be modified
E_SE_DRLC_HEATING_TEMPERATURE_SET_PC	Specifies that 'heating temperature set-point' is to be modified
E_SE_DRLC_AVERAGE_LOAD_ADJUSTMENT_PI	Specifies that 'average load adjustment percentage' is to be modified
E_SE_DRLC_DUTY_CYCLE_APPLIED	Specifies that 'duty cycle' is to be modified

Parent topic: Enumerations

'LCE List' Enumerations The LCE lists are enumerated in the teSE_DRLCEventList structure below:

```
typedef enum PACK
{
    E_SE_DRLC_EVENT_LIST_SCHEDULED =0x00,
    E_SE_DRLC_EVENT_LIST_ACTIVE,
    E_SE_DRLC_EVENT_LIST_CANCELLED,
    E_SE_DRLC_EVENT_LIST_DEALLOCATED,
    E_SE_DRLC_EVENT_LIST_NONE
} teSE_DRLCEventList;
```

The above lists are described in the table below.

LCE List Enumeration	Description
E_SE_DRLC_EVENT_LI	**Scheduled list:** Contains LCEs that are due to be executed in the future
E_SE_DRLC_EVENT_LI	**Active list:** Contains LCEs that are currently being executed
E_SE_DRLC_EVENT_LI	**Cancelled list:** Contains LCEs that have been cancelled with a randomized end-time and whose random end-time has not yet been reached
E_SE_DRLC_EVENT_LI	**Deallocated list:** Contains expired LCEs and therefore a record of the free storage for LCEs

Parent topic: Enumerations

'LCE Status' Enumerations LCE status is enumerated in the teSE_DRLCEventStatus structure below:

```
typedef enum PACK
{
    E_SE_DRLC_LOAD_CONTROL_EVENT_COMMAND_RECIEVED =0x01,
    E_SE_DRLC_EVENT_STARTED,
    E_SE_DRLC_EVENT_COMPLETED,
    E_SE_DRLC_USER_CHOSEN_OPT_OUT,
    E_SE_DRLC_USER_CHOSEN_OPT_IN,
    E_SE_DRLC_EVENT_HAS_BEEN_CANCELLED,
    E_SE_DRLC_EVENT_HAS_BEEN_SUPERSEDED,
    E_SE_DRLC_EVENT_PARTIALLY_COMPLETED_WITH_USER_OPT_OUT,
    E_SE_DRLC_EVENT_PARTIALLY_COMPLETED_WITH_USER_OPT_IN,
    E_SE_DRLC_EVENT_COMPLETED_NO_USER_PARTICIPATION,
```

(continues on next page)

(continued from previous page)

```
    E_SE_DRLC_REJECTED_INVALID_CANCEL_COMMAND_DEFAULT =0xF8,
    E_SE_DRLC_REJECTED_INVALID_CANCEL_COMMAND_INVALID_EFFECTIVE_TIME,
    E_SE_DRLC_REJECTED_EVENT_RECEIVED_AFTER_IT_HAD_EXPIRED =0xFB,
    E_SE_DRLC_REJECTED_INVALID_CANCEL_COMMAND_UNDEFINED_EVENT=0xFD,
    E_SE_DRLC_LOAD_CONTROL_EVENT_COMMAND_REJECTED,
    E_SE_DRLC_EVENT_STATUS_ENUM_END
} teSE_DRLCEventStatus;
```

The above enumerations are described in the table below.

Enumeration	Description
E_SE_DRLC_LOAD_CONTROL_EVENT_COMMAND_RECEIVED	LCE command received (to add new LCE to local lists)
E_SE_DRLC_EVENT_STARTED	LCE has started
E_SE_DRLC_EVENT_COMPLETED	LCE has completed
E_SE_DRLC_USER_CHOSEN_OPT_OUT	Client has opted out of the LCE
E_SE_DRLC_USER_CHOSEN_OPT_IN	Client has opted into the LCE
E_SE_DRLC_EVENT_HAS_BEEN_CANCELLE	LCE has been cancelled
E_SE_DRLC_EVENT_HAS_BEEN_SUPERSEDI	LCE has been replaced with another LCE
E_SE_DRLC_EVENT_PARTIALLY_COMPLETI	LCE has prematurely completed due to a client opt-out during the LCE
E_SE_DRLC_EVENT_PARTIALLY_COMPLETI	LCE has completed but was only partially executed due to a client opt-in during the LCE
E_SE_DRLC_EVENT_COMPLETED_NO_USER	LCE has completed but there was no client participation (due to a client opt-out from the start)
E_SE_DRLC_REJECTED_INVALID_CANCEL_C	Received 'cancel command' invalid and rejected (default)
E_SE_DRLC_REJECTED_INVALID_CANCEL_C	Received 'cancel command' rejected due to invalid effective time (start-time of cancellation)
E_SE_DRLC_REJECTED_EVENT_RECEIVED_A	LCE was received after it had expired (current time is greater than start-time + duration)
E_SE_DRLC_REJECTED_INVALID_CANCEL_C	Received 'cancel command' due to undefined LCE
E_SE_DRLC_LOAD_CONTROL_EVENT_COMMAND_REJECTED	LCE command rejected

Parent topic: Enumerations

Parent topic:*Demand-Response and Load Control Cluster*

Structures

tsSE_DRLCLoadControlEvent The structure of type tsSE_DRLCLoadControlEvent contains the parameters of a Load Control Event (LCE), as shown and described below.

```
typedef struct {
 uint8    u8UtilityEnrolmentGroup;
 uint8    u8CriticalityLevel;
 uint8    u8CoolingTemperatureOffset;
 uint8    u8HeatingTemperatureOffset;
 uint8    u8AverageLoadAdjustmentSetPoint;
 uint8    u8DutyCycle;
 uint8    u8EventControl;
 uint16   u16DeviceClass;
```

(continues on next page)

(continued from previous page)

```
 uint16   u16DurationInMinutes;
 uint16   u16CoolingTemperatureSetPoint;
 uint16   u16HeatingTemperatureSetPoint;
 uint32   u32IssuerId;
 uint32   u32StartTime;
}tsSE_DRLCLoadControlEvent;
```

where:

- u8UtilityEnrolmentGroup identifies the group of devices to which the LCE applies - an 'enrolment group' is defined by the utility company. The identifier 0x00 is reserved to indicate 'all groups'.
- u8CriticalityLevel is a value representing the level of criticality of the LCE. Enumerations are provided for the different levels and are detailed in Section 41.10.3.
- u8CoolingTemperatureOffset (optional) specifies the required temperature offset, in units of 0.1oC, above the current temperature set-point of a cooling device (example, 0x5 represents a temperature offset of 0.5oC). The setting 0xFF is used to indicate that no offset is required.
- u8HeatingTemperatureOffset (optional) specifies the required temperature offset, in units of 0.1oC, below the current temperature set-point of a heating device (example, 0x14 represents a temperature offset of 2.0oC). The setting 0xFF is used to indicate that no offset is required.
- u8AverageLoadAdjustmentSetPoint (optional) specifies the maximum permissible load as an offset from the consumer's average load, where this offset is expressed as a positive or negative percentage in units of 1% (e.g. 20% allows loads of up to 120% of the average while -10% allows loads of up to 90% of the average). The offset has a valid range of -100% to +100% and is represented in two's complement form (e.g. 15% is represented by 0x0F and

  -5% is represented by 0xFB). The value 0x80 is used to indicate that no such limit is required.
- u8DutyCycle (optional) specifies the percentage duty cycle for the load supplied to the device - that is, the percentage of the LCE duration for which the load will be supplied (e.g. for a duty cycle of 80%, the supplied device will be 'on' for 80% of the duration of the LCE). The manner in which the duty cycle is implemented (e.g. periodicity) is device-specific. The valid range of duty cycle values is 0 to 100. The setting 0xFF indicates that no duty cycling is required.
- u8EventControl specifies whether a randomised start-time and/or randomised end-time are required for the LCE. The following bit-masks are provided to allow the start-time and end-time of an LCE to be individually randomised (they can be bitwise-ORed to randomize both times):

```
#define SE_DRLC_CONTROL_RANDOMISATION_START_TIME_MASK   (0x01)
#define SE_DRLC_CONTROL_RANDOMISATION_STOP_TIME_MASK (0x02)
```

- u``16DeviceClass identifies the class(es) of device to which the LCE applies. Enumerations are provided for the various device classes, which may be combined in a bitwise-OR operation, and are detailed in Section 41.10.1.
- u16DurationInMinutes specifies the duration, in minutes, of the LCE (although the actual duration will be longer than the specified duration if a randomized end-time is required). The maximum possible duration that can be specified is 1440 minutes (one day).
- u16CoolingTemperatureSetPoint (optional) specifies the required temperature set-point, in units of 0.01oC, for a cooling device, where a negative temperature is represented in two's complement form (e.g. a temperature of 20oC is represented by 0x07D0 and -40oC is repre-

sented by 0xF060). The valid temperature range is -273.15°C to 327.67°C. The setting 0x8000 is used to indicate that no temperature set-point is required.

- u16HeatingTemperatureSetPoint (optional) specifies the required temperature set-point, in units of 0.01oC, for a heating device, where a negative temperature is represented in two's complement form (for example, a temperature of 25oC is represented by 0x09C4 and -1oC is represented by 0xFFFF). The valid temperature range is -273.15°C to 327.67°C. The setting 0x8000 is used to indicate that no temperature set-point is required.
- u32IssuerId is a unique identifier for the LCE, issued by the utility company (the value could be based on the time-stamp of when the LCE was issued).
- u``32StartTime represents the start-time (UTC) of the LCE (although the actual start-time is later if a randomized start-time is required). The value 0x000000000 is used to indicate a 'start-time of now'.

Parent topic: Structures

tsSE_DRLCGetScheduledEvents The structure of type tsSE_DRLCGetScheduledEvents contains the parameters of a **Get Scheduled Event** message, as shown and described below.

```
typedef struct {
   uint32 u32StartTime;
   uint8 u8numberOfEvents;
} tsSE_DRLCGetScheduledEvents;
```

where:

- u32StartTime is the earliest start-time (UTC) of the requested LCEs
- u8numberOfEvents is the maximum number of LCEs to report

Parent topic: Structures

tsSE_DRLCCancelLoadControlEvent The structure of type tsSE_DRLCCancelLoadControlEvent contains the parameters of a Cancel LCE command, as shown and described below.

```
typedef struct {
    uint32    u32IssuerId;
    uint16    u16DeviceClass;
    uint8     u8UtilityEnrolmentGroup;
    teSE_DRLCCancelControl  eCancelControl;
    uint32    u32effectiveTime;
} tsSE_DRLCCancelLoadControlEvent;
```

where:

- u32IssuerId is the identifier (provided by the utility company) of the LCE to be canceled
- u16DeviceClass is a bitmap indicating the device class(es) to which the LCE cancelation applies - enumerations for the device classes are provided, as described in Section 41.10.1
- u8UtilityEnrolmentGroup is the enrolment group of the devices to which the LCE cancelation applies
- eCancelControl indicates whether to honour a randomised end that has been configured in the LCE - enumerations are provided, as described in Section 41.10.4
- u32effectiveTime is the time (UTC) from which the LCE cancelation is effective

Parent topic: Structures

tsSE_DRLCReportEvent The structure of type tsSE_DRLCReportEvent contains the parameters of a Report Event Status message, as shown and described below.

```
typedef struct {
    uint8    u8EventStatus;
    uint8    u8AverageLoadAdjustmentPercentageApplied;
    uint8    u8DutyCycleApplied;
    uint8    u8EventControl;
    uint8    u8SignatureType;
    uint8    u8CriticalityLevelApplied;
    bool_t   bSignatureVerified;
    uint16   u16CoolingTemperatureSetPointApplied;
    uint16   u16HeatingTemperatureSetPointApplied;
    uint32   u32IssuerId;
    uint32   u32EventStatusTime;
    tsSE_DRLCOctets      sSignature;
} tsSE_DRLCReportEvent;
```

where:

- u8EventStatus is the reported LCE status - enumerations are provided and described in Section 41.10.8
- u8AverageLoadAdjustmentPercentageApplied is an optional field containing the load adjustment percentage applied by the sending client (if the user has chosen to over-ride the original setting in the LCE) - for the format of this setting, refer to the equivalent field description in Section 41.11.1 (0x80 indicates that the field is not used)
- u8DutyCycleApplied is an optional field containing the percentage duty cycle applied by the sending client (if the user has chosen to over-ride the original setting in the LCE) - for the format of this setting, refer to the equivalent field description in Section 41.11.1 (0xFF indicates that the field is not used)
- u8EventControl is a bitmap which specifies whether a randomised start-time and/or randomised end-time are configured for the LCE:

Bit	Description
0	1 = randomised start-time, 0 = immediate start-time
1	1 = randomised end-time, 0 = immediate end-time
2-7	Not used

- u8SignatureType is the type of algorithm, if any, used to create the signature for the Report Event Status message (only one algorithm, ECDSA, is currently supported):

```
#define SE_DRLC_NO_SIGNATURE (0x00)
#define SE_DRLC_SIGNATURE_TYPE_ECDSA      (0x01)
```

- u8CriticalityLevelApplied is the criticality level of the LCE - enumerations are provided and described in Section 41.10.3
- bSignatureVerified is filled in by the recipient of the Report Event Status message (therefore, the DRLC cluster server) to indicate whether the signature of the message has been verified and is valid:

  TRUE - verified and valid

  FALSE - verified and not valid, or not verified
- u16CoolingTemperatureSetPointApplied is an optional field containing the cooling temperature applied by the sending client (if the user has chosen to over-ride the original setting in

the LCE) - for the format of this setting, refer to the equivalent field description in Section 41.11.1 (0x8000 indicates that the field is not used)

- u16HeatingTemperatureSetPointApplied is an optional field containing the heating temperature applied by the sending client (if the user has chosen to over-ride the original setting in the LCE) - for the format of this setting, refer to the equivalent field description in Section 41.11.1 (0x8000 indicates that the field is not used)
- u32IssuerId is the unique identifier for the LCE, as issued by the utility company
- u``32EventStatusTime is the time (UTC) at which the Report Event Status message was issued
- s``Signature is the signature for the Report Event Status message - this is the concatenation of two ECDSA signature components (r,s)

Note: It is recommended that signatures are supported by your application for backward compatibility.

Parent topic: Structures

tsSE_DRLCCallBackMessage The structure of type tsSE_DRLCCallBackMessage contains a DRLC callback event. It is shown below but described in Section 41.7.

```
typedef struct
{
  teSE_DRLCCallBackEventType  eEventType;
  uint8 u8CommandId;
  teSE_DRLCStatus  eDRLCStatus;
  uint32      u32CurrentTime;
  union {
   tsSE_DRLCLoadControlEvent       sLoadControlEvent;
   tsSE_DRLCCancelLoadControlEvent      sCancelLoadControlEvent;
   tsSE_DRLCCancelLoadControlAllEvent  sCancelLoadControlAllEvent;
   tsSE_DRLCReportEvent     sReportEvent;
   tsSE_DRLCGetScheduledEvents     sGetScheduledEvents;
    } uMessage;
} tsSE_DRLCCallBackMessage;
```

Parent topic: Structures

Parent topic:*Demand-Response and Load Control Cluster*

Compile-time options This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the DRLC cluster.

The DRLC cluster is enabled by defining CLD_DRLC.

Client and server versions of the cluster are defined by DRLC_CLIENT and DRLC_SERVER, respectively.

Length of LCE Lists The number of LCEs that may be stored in an LCE list (see Section 41.4.2) is, by default, three. This default can be over-ridden on the cluster server and a cluster client by assigning the desired values to the macros:

SE_DRLC_NUMBER_OF_SERVER_LOAD_CONTROL_ENTRIES (server)

SE_DRLC_NUMBER_OF_CLIENT_LOAD_CONTROL_ENTRIES (client)

LCE Re-sends The DRLC cluster server may re-send an LCE when it becomes active in order to support clients that do not have a clock. This facility should not be enabled unless explicitly required. To enable this functionality, define:

DRLC_SEND_LCE_AGAIN_AT_ACTIVE_TIME

Message Signing (Security) On DRLC cluster clients that need to implement message signing (see Section 41.6), the following must be defined:

```
#define SE_MESSAGE_SIGNING
```

For a DRLC cluster server to check a message signature, it is necessary to locally store the certificates of any nodes that perform key establishment. The maximum number of certificates that can be stored is configured by defining the following on the server:

```
#define KEC_NUM_CERTIFICATES <      n>
```

where n is the number of certificates that can be stored.

Parent topic:*Demand-Response and Load Control Cluster*

Simple Metering Cluster This chapter outlines the Simple Metering cluster, which is used to handle information relating to the measured consumption of some resource, which may be electricity, gas, heat or water.

The Simple Metering cluster has a Cluster ID of 0x0702.

Overview The Simple Metering cluster is required in ZigBee devices as indicated in the table below.

	Server-side	Client-side
Mandatory in...	Metering Device	
Optional in...	ESP	ESP

IPD

PCT

|

Thus, a Metering Device or ESP can use this cluster to store attributes and respond to commands relating to these attributes. An IPD or PCT may use this cluster to issue commands to interact with remote attributes held on a Metering Device or ESP.

The Simple Metering cluster is enabled by defining CLD_SIMPLE_METERING in the **zcl_options.h** file. Further compile-time options for the Simple Metering cluster are detailed in Section 42.12.

The information that can potentially be stored in this cluster is organized into the following attribute sets:

- Reading Information Set (resource measurement information)
- TOU Information Set (Time-Of-Use information)
- Meter Status
- Formatting (data formatting/interpretation guidance)

- Historical Consumption
- Load Profile Configuration
- Supply Limit
- Block Information (for future use - not certifiable in SE 1.1.1 or earlier)
- Alarms (for future use - not certifiable in SE 1.1.1 or earlier)

This information is stored in both mandatory and optional attributes - see Section 42.3.

Note: Many of the Simple Metering cluster attributes are not certifiable in SE 1.1.1 (07-5356-17) or earlier and are reserved for future use (as indicated in Section 42.2).

Parent topic:*Simple Metering Cluster*

Simple Metering Cluster structure and attributes The Simple Metering cluster is contained in the following tsSE_SimpleMetering structure:

```
typedef struct
{
    /* Reading information attribute set attribute ID's (D.3.2.2.1) */
    zuint48          u48CurrentSummationDelivered;        /
* Mandatory */
#ifdef CLD_SM_ATTR_CURRENT_SUMMATION_RECEIVED
    zuint48          u48CurrentSummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_MAX_DEMAND_DELIVERED
    zuint48          u48CurrentMaxDemandDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_MAX_DEMAND_RECEIVED
    zuint48          u48CurrentMaxDemandReceived;
#endif
#ifdef CLD_SM_ATTR_DFT_SUMMATION
    zuint48          u48DFTSummation;
#endif
#ifdef CLD_SM_ATTR_DAILY_FREEZE_TIME
    zuint16          u16DailyFreezeTime;
#endif
#ifdef CLD_SM_ATTR_POWER_FACTOR
    zint8            i8PowerFactor;
#endif
#ifdef CLD_SM_ATTR_READING_SNAPSHOT_TIME
    zutctime          utctReadingSnapshotTime;
#endif
#ifdef CLD_SM_ATTR_CURRENT_MAX_DEMAND_DELIVERED_TIME
    zutctime          utctCurrentMaxDemandDeliveredTime;
#endif
#ifdef CLD_SM_ATTR_CURRENT_MAX_DEMAND_RECEIVED_TIME
    zutctime          utctCurrentMaxDemandReceivedTime;
#endif
#ifdef CLD_SM_ATTR_DEFAULT_UPDATE_PERIOD
    uint8            u8DefaultUpdatePeriod;
#endif
#ifdef CLD_SM_ATTR_FAST_POLL_UPDATE_PERIOD
    uint8            u8FastPollUpdatePeriod;
#endif
#ifdef CLD_SM_ATTR_CURRENT_BLOCK_PERIOD_CONSUMPTION_DELIVERED
    zuint48          u48CurrentBlockPeriodConsumptionDelivered;
#endif
#ifdef CLD_SM_ATTR_DAILY_CONSUMPTION_TARGET
    zuint24          u24DailyConsumptionTarget;
#endif
```

(continues on next page)

(continued from previous page)

```
#ifdef CLD_SM_ATTR_CURRENT_BLOCK
    zenum8          e8CurrentBlock;
#endif
#ifdef CLD_SM_SUPPORT_GET_PROFILE
#ifdef CLD_SM_ATTR_PROFILE_INTERVAL_PERIOD
    zenum8          eProfileIntervalPeriod;
#endif
#ifdef CLD_SM_ATTR_INTERVAL_READ_REPORTING_PERIOD
    uint16          u16IntervalReadReportingPeriod;
#endif
#endif  // CLD_SM_SUPPORT_GET_PROFILE
#ifdef CLD_SM_ATTR_PREVIOUS_BLOCK_PERIOD_CONSUMPTION_DELIVERED
    zuint48         u48PreviousBlockPeriodConsumptionDelivered;
#endif
#ifdef CLD_SM_ATTR_PRESET_READING_TIME
    uint16          u16PresetReadingTime;
#endif
#ifdef CLD_SM_ATTR_VOLUME_PER_REPORT
    uint16          u16VolumePerReport;
#endif
#ifdef CLD_SM_ATTR_FLOW_RESTRICTION
    uint8           u8FlowRestriction;
#endif
#ifdef CLD_SM_ATTR_SUPPLY_STATUS
    zbmap8          u8SupplyStatus;
#endif
#ifdef CLD_SM_ATTR_CURRENT_INLET_ENERGY_CARRIER_SUMMATION
    zuint48         u48CurrentInletEnergyCarrierSummation;
#endif
#ifdef CLD_SM_ATTR_CURRENT_OUTLET_ENERGY_CARRIER_SUMMATION
    zuint48         u48CurrentOutletEnergyCarrierSummation;
#endif
#ifdef CLD_SM_ATTR_INLET_TEMPERATURE
    int16           i16InletTemperature;
#endif
#ifdef CLD_SM_ATTR_OUTLET_TEMPERATURE
    int16           i16OutletTemperature;
#endif
#ifdef CLD_SM_ATTR_CONTROL_TEMPERATURE
    int16           i16ControlTemperature;
#endif
#ifdef CLD_SM_ATTR_CURRENT_INLET_ENERGY_CARRIER_DEMAND
    zint24          i24CurrentInletEnergyCarrierDemand;
#endif
#ifdef CLD_SM_ATTR_CURRENT_OUTLET_ENERGY_CARRIER_DEMAND
    zint24          i24CurrentOutletEnergyCarrierDemand;
#endif
    /* Time Of Use Information attribute attribute ID's set (D.3.2.2.2) */
#ifdef CLD_SM_ATTR_CURRENT_TIER_1_SUMMATION_DELIVERED
    zuint48         u48CurrentTier1SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_1_SUMMATION_RECEIVED
    zuint48         u48CurrentTier1SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_2_SUMMATION_DELIVERED
    zuint48         u48CurrentTier2SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_2_SUMMATION_RECEIVED
    zuint48         u48CurrentTier2SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_3_SUMMATION_DELIVERED
```

(continues on next page)

(continued from previous page)

```
    zuint48          u48CurrentTier3SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_3_SUMMATION_RECEIVED
    zuint48          u48CurrentTier3SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_4_SUMMATION_DELIVERED
    zuint48          u48CurrentTier4SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_4_SUMMATION_RECEIVED
    zuint48          u48CurrentTier4SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_5_SUMMATION_DELIVERED
    zuint48          u48CurrentTier5SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_5_SUMMATION_RECEIVED
    zuint48          u48CurrentTier5SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_6_SUMMATION_DELIVERED
    zuint48          u48CurrentTier6SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_6_SUMMATION_RECEIVED
    zuint48          u48CurrentTier6SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_7_SUMMATION_DELIVERED
    zuint48          u48CurrentTier7SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_7_SUMMATION_RECEIVED
    zuint48          u48CurrentTier7SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_8_SUMMATION_DELIVERED
    zuint48          u48CurrentTier8SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_8_SUMMATION_RECEIVED
    zuint48          u48CurrentTier8SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_9_SUMMATION_DELIVERED
    zuint48          u48CurrentTier9SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_9_SUMMATION_RECEIVED
    zuint48          u48CurrentTier9SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_10_SUMMATION_DELIVERED
    zuint48          u48CurrentTier10SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_10_SUMMATION_RECEIVED
    zuint48          u48CurrentTier10SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_11_SUMMATION_DELIVERED
    zuint48          u48CurrentTier11SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_11_SUMMATION_RECEIVED
    zuint48          u48CurrentTier11SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_12_SUMMATION_DELIVERED
    zuint48          u48CurrentTier12SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_12_SUMMATION_RECEIVED
    zuint48          u48CurrentTier12SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_13_SUMMATION_DELIVERED
    zuint48          u48CurrentTier13SummationDelivered;
```

(continues on next page)

(continued from previous page)

```
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_13_SUMMATION_RECEIVED
    zuint48          u48CurrentTier13SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_14_SUMMATION_DELIVERED
    zuint48          u48CurrentTier14SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_14_SUMMATION_RECEIVED
    zuint48          u48CurrentTier14SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_15_SUMMATION_DELIVERED
    zuint48          u48CurrentTier15SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_15_SUMMATION_RECEIVED
    zuint48          u48CurrentTier15SummationReceived;
#endif
    /* Meter status attribute set attribute ID's (D.3.2.2.3) */
    zbmap8           u8MeterStatus;                    /* Mandatory */
#ifdef CLD_SM_ATTR_REMAINING_BATTERY_LIFE
    uint8            u8RemainingBatteryLife;
#endif
#ifdef CLD_SM_ATTR_HOURS_IN_OPERATION
    zuint24          u24HoursInOperation;
#endif
#ifdef CLD_SM_ATTR_HOURS_IN_FAULT
    zuint24          u24HoursInFault;
#endif
    /* Formatting attribute set attribute ID's (D.3.2.2.4) */
    zenum8           eUnitOfMeasure;                   /* Mandatory */
#ifdef CLD_SM_ATTR_MULTIPLIER
    zuint24          u24Multiplier;
#endif
#ifdef CLD_SM_ATTR_DIVISOR
    zuint24          u24Divisor;
#endif
    zbmap8           u8SummationFormatting;            /* Mandatory */
#ifdef CLD_SM_ATTR_DEMAND_FORMATING
    zbmap8           u8DemandFormatting;
#endif
#ifdef CLD_SM_ATTR_HISTORICAL_CONSUMPTION_FORMATTING
    zbmap8           u8HistoricalConsumptionFormatting;
#endif
    zbmap8           eMeteringDeviceType;              /* Mandatory */
#ifdef CLD_SM_ATTR_SITE_ID
    tsZCL_OctetString  sSiteId;
    uint8            au8SiteId[SE_SM_SITE_ID_MAX_STRING_LENGTH];
#endif
#ifdef CLD_SM_ATTR_METER_SERIAL_NUMBER
   tsZCL_OctetString  sMeterSerialNumber;
 uint8   au8MeterSerialNumber[SE_SM_METER_SERIAL_NUMBER_MAX_STRING_LENGTH];
#endif
#ifdef CLD_SM_ATTR_ENERGY_CARRIER_UNIT_OF_MEASURE
    zenum8         e8EnergyCarrierUnitOfMeasure;
#endif
#ifdef CLD_SM_ATTR_ENERGY_CARRIER_SUMMATION_FORMATTING
    zbmap8         u8EnergyCarrierSummationFormatting;
#endif
#ifdef CLD_SM_ATTR_ENERGY_CARRIER_DEMAND_FORMATTING
    zbmap8         u8EnergyCarrierDemandFormatting;
#endif
#ifdef CLD_SM_ATTR_TEMPERATURE_UNIT_OF_MEASURE
```

(continues on next page)

(continued from previous page)

```
    zenum8          e8TemperatureUnitOfMeasure;
#endif
#ifdef CLD_SM_ATTR_TEMPERATURE_FORMATTING
    zbmap8          u8TemperatureFormatting;
#endif
    /* ESP Historical Consumption set attribute ID's (D.3.2.2.5) */
#ifdef CLD_SM_ATTR_INSTANTANEOUS_DEMAND
    zint24          i24InstantaneousDemand;
#endif
#ifdef CLD_SM_ATTR_CURRENT_DAY_CONSUMPTION_DELIVERED
    zuint24         u24CurrentDayConsumptionDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_DAY_CONSUMPTION_RECEIVED
    zuint24         u24CurrentDayConsumptionReceived;
#endif
#ifdef CLD_SM_ATTR_PREVIOUS_DAY_CONSUMPTION_DELIVERED
    zuint24         u24PreviousDayConsumptionDelivered;
#endif
#ifdef CLD_SM_ATTR_PREVIOUS_DAY_CONSUMPTION_RECEIVED
    zuint24         u24PreviousDayConsumptionReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_DELIVERED
 zutctime      utctCurrentPartialProfileIntervalStartTimeDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_RECEIVED
 zutctime      utctCurrentPartialProfileIntervalStartTimeReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_DELIVERED
 zuint24   u24CurrentPartialProfileIntervalValueDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_RECEIVED
 zuint24    u24CurrentPartialProfileIntervalValueReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_DAY_MAXIMUM_PRESSURE
 zuint48   u48CurrentDayMaxPressure;
#endif
#ifdef CLD_SM_ATTR_CURRENT_DAY_MINIMUM_PRESSURE
 zuint48     u48CurrentDayMinPressure;
#endif
#ifdef CLD_SM_ATTR_PREVIOUS_DAY_MAXIMUM_PRESSURE
  zuint48   u48PreviousDayMaxPressure;
#endif
#ifdef CLD_SM_ATTR_PREVIOUS_DAY_MINIMUM_PRESSURE
   zuint48      u48PreviousDayMinPressure;
#endif
#ifdef CLD_SM_ATTR_CURRENT_DAY_MAXIMUM_DEMAND
  zint24    i24CurrentDayMaxDemand;
#endif
#ifdef CLD_SM_ATTR_PREVIOUS_DAY_MAXIMUM_DEMAND
  zint24       i24PreviousDayMaxDemand;
#endif
#ifdef CLD_SM_ATTR_CURRENT_MONTH_MAXIMUM_DEMAND
  zint24       i24CurrentMonthMaxDemand;
#endif
#ifdef CLD_SM_ATTR_CURRENT_YEAR_MAXIMUM_DEMAND
    zint24      i24CurrentYearMaxDemand;
#endif
#ifdef CLD_SM_ATTR_CURRENT_DAY_MAXIMUM_ENERGY_CARRIER_DEMAND
  zint24          i24CurrentDayMaxEnergyCarrierDemand;
#endif
#ifdef CLD_SM_ATTR_PREVIOUS_DAY_MAXIMUM_ENERGY_CARRIER_DEMAND
```

(continues on next page)

(continued from previous page)

```
  zint24          i24PreviousDayMaxEnergyCarrierDemand;
#endif
#ifdef CLD_SM_ATTR_CURRENT_MONTH_MAXIMUM_ENERGY_CARRIER_DEMAND
  zint24          i24CurrentMonthMaxEnergyCarrierDemand;
#endif
#ifdef CLD_SM_ATTR_CURRENT_MONTH_MINIMUM_ENERGY_CARRIER_DEMAND
  zint24          i24CurrentMonthMinEnergyCarrierDemand;
#endif
#ifdef CLD_SM_ATTR_CURRENT_YEAR_MAXIMUM_ENERGY_CARRIER_DEMAND
  zint24          i24CurrentYearMaxEnergyCarrierDemand;
#endif
#ifdef CLD_SM_ATTR_CURRENT_YEAR_MINIMUM_ENERGY_CARRIER_DEMAND
  zint24          i24CurrentYearMinEnergyCarrierDemand;
#endif
/* Load Profile attribute set attribute ID's (D.3.2.2.6) */
#ifdef CLD_SM_ATTR_MAX_NUMBER_OF_PERIODS_DELIVERED
  zuint8    u8MaxNumberOfPeriodsDelivered;
#endif
/* Supply Limit attribute set attribute ID's (D.3.2.2.7) */
#ifdef CLD_SM_ATTR_CURRENT_DEMAND_DELIVERED
 zuint24       u24CurrentDemandDelivered;
#endif
#ifdef CLD_SM_ATTR_DEMAND_LIMIT
    zuint24     u24DemandLimit;
#endif
#ifdef CLD_SM_ATTR_DEMAND_INTEGRATION_PERIOD
    zuint8       u8DemandIntegrationPeriod;
#endif
#ifdef CLD_SM_ATTR_NUMBER_OF_DEMAND_SUBINTERVALS
    zuint8       u8NumberOfDemandSubintervals;
#endif
/* Block Information attribute set attribute ID's (D.3.2.2.8) */
/* No Tier Block */
#if (CLD_SM_ATTR_NO_TIER_BLOCK_CURRENT_SUMMATION_DELIVERED_MAX_COUNT !
↪= 0)
 zuint48  au48CurrentNoTierBlockSummationDelivered
[CLD_SM_ATTR_NO_TIER_BLOCK_CURRENT_SUMMATION_DELIVERED_MAX_COUNT];
#endif
    /* Tier 1 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 0)&&
 (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED !
↪= 0))
 zuint48  au48CurrentTier1BlockSummationDelivered
   [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif
    /* Tier 2 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 1)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED != 0))
zuint48  au48CurrentTier2BlockSummationDelivered
   [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif
/* Tier 3 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 2)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED != 0))
  zuint48    au48CurrentTier3BlockSummationDelivered
    [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif
    /* Tier 4 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 3)&&
```

(continues on next page)

(continued from previous page)

```
    (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED != 0))
   zuint48   au48CurrentTier4BlockSummationDelivered
   [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif
   /* Tier 5 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 4)&&
    (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED != 0))
   zuint48   au48CurrentTier5BlockSummationDelivered
   [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif
   /* Tier 6 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 5)&&
    (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED != 0))
   zuint48      au48CurrentTier6BlockSummationDelivered
    [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED];
#endif
   /* Tier 7 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 6)&&
    (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED != 0))
   zuint48      au48CurrentTier7BlockSummationDelivered
    [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED];
#endif
   /* Tier 8 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 7)&&
    (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED != 0))
   zuint48      au48CurrentTier8BlockSummationDelivered
    [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED];
#endif
   /* Tier 9 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 8)&&
    (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED != 0))
   zuint48      au48CurrentTier9BlockSummationDelivered
     [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED];
#endif
   /* Tier 10 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 9)&&
    (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED != 0))
   zuint48     au48CurrentTier10BlockSummationDelivered
    [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED];
#endif
   /* Tier 11 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 10)&&
    (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED != 0))
   zuint48  au48CurrentTier11BlockSummationDelivered
     [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED];
#endif
   /* Tier 12 Block Set */
```

(continues on next page)

(continued from previous page)

```
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 11)&&
    (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED != 0))
   zuint48   au48CurrentTier12BlockSummationDelivered
     [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED];
#endif
   /* Tier 13 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 12)&&
    (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED != 0))
   zuint48   au48CurrentTier13BlockSummationDelivered
     [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED];
#endif
   /* Tier 14 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 13)&&
  CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED !
↪= 0))
  zuint48     au48CurrentTier14BlockSummationDelivered
   [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif
   /* Tier 15 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 14)&&
   (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED !
↪= 0))
   zuint48  au48CurrentTier15BlockSummationDelivered
    [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_
↪DELIVERED];
#endif
   /* Alarm attribute set attribute ID's (D.3.2.2.9) */
#ifdef CLD_SM_ATTR_GENERIC_ALARM_MASK
   zbmap16     u16GenericAlarmMask;
#endif
#ifdef CLD_SM_ATTR_ELECTRICITY_ALARM_MASK
   zbmap32     u32ElectricityAlarmMask;
#endif
#ifdef CLD_SM_ATTR_PRESSURE_ALARM_MASK
   zbmap16      u16PressureAlarmMask;
#endif
#ifdef CLD_SM_ATTR_WATER_SPECIFIC_ALARM_MASK
   zbmap16      u16WaterSpecificAlarmMask;
#endif
#ifdef CLD_SM_ATTR_HEAT_AND_COOLING_ALARM_MASK
   zbmap16      u16HeatAndCoolingSpecificAlarmMask;
#endif
#ifdef CLD_SM_ATTR_GAS_ALARM_MASK
   zbmap16      u16GasAlarmMask;
#endif
} tsCLD_SimpleMetering;
```

where:

'Reading Information' Attribute Set

- u48CurrentSummationDelivered is the total amount of the measured resource (e.g. electrical energy) delivered to the premises so far, expressed in the units specified in eUnitOfMeasure and in the format specified in u8SummationFormatting
- The following are optional attributes and are fully described in the *ZigBee Smart Energy*

Profile Specification:

```
u48CurrentSummationReceived
u48CurrentMaxDemandDelivered
u48CurrentMaxDemandReceived
u48DFTSummation
u16DailyFreezeTime
i8PowerFactor
utctReadingSnapshotTime
utctCurrentMaxDemandDeliveredTime
utctCurrentMaxDemandReceivedTime
```

- The following are optional attributes that relate to Fast Polling mode (*both attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - u32DefaultUpdatePeriod is the default poll-period, in seconds, that is used in updating metering data outside of fast polling episodes
  - u8FastPollUpdatePeriod is the minimum poll-period, in seconds, that can be used in updating metering data during fast polling episodes (should not be set to less than 2 seconds)
- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*(*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):

```
u48CurrentBlockPeriodConsumptionDelivered
u24DailyConsumptionTarget
e8CurrentBlock
```

- The following are optional attributes that relate to the 'Get Profile' feature:
  - eProfileIntervalPeriod is the time-interval over which one set of consumption data will be collected
  - u32IntervalReadReportingPeriod is the time-interval, in minutes, after which a sleepy End Device should wake up to provide metering data
- The following are optional attributes are fully described in the *ZigBee Smart Energy Profile Specification* (*all these attributes except*u8SupplyStatus *are not certifiable in SE 1.1.1 or earlier and are for future use*):

```
u16PresetReadingTime
u16VolumePerReport
u8FlowRestriction
u8SupplyStatus
u48CurrentInletEnergyCarrierSummation
u48CurrentOutletEnergyCarrierSummation
i16InletTemperature
i16OutletTemperature
i16ControlTemperature
i24CurrentInletEnergyCarrierDemand
i24CurrentOutletEnergyCarrierDemand
```

'Time-Of-Use (TOU) Information' Attribute Set

- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*(*the attributes for tiers 7 to 15 are not certifiable in SE 1.1.1 or earlier and are for future use*):

```
u48CurrentTier1SummationDelivered
u48CurrentTier1SummationReceived
u48CurrentTier2SummationDelivered
u48CurrentTier2SummationReceived
...
...
...
...
u48CurrentTier15SummationDelivered
u48CurrentTier15SummationReceived
```

```
```

'Meter Status' Attribute Set

- u``8MeterStatus is an 8-bit bitmap representing the status of the meter. Enumerated masks are provided that correspond to the possible settings - see Section 42.10.2*(this attribute is only certifiable for electricity meters in SE 1.1.1)*
- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*(*all the attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):

```
u8RemainingBatteryLife
u24HoursInOperation
u24HoursInFault
```

'Formatting' Attribute Set

- eUnitOfMeasure indicates the unit of measure for the resource quantity contained above in u48CurrentSummationDelivered and below in i24InstantaneousDemand. Enumerations for the possible units are provided - see Section 42.10.3
- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*:

```
u24Multiplier
u24Divisor
```

- u8SummationFormatting indicates the formatting for the resource quantity contained above in u48CurrentSummationDelivered. Enumerations for the possible formats are provided - see Section 42.10.4
- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*:

```
u8DemandFormatting
u8HistoricalConsumptionFormatting
```

- eMeteringDeviceType indicates the type of Metering Device in terms of the resource type which it measures. Enumerations for the possible device types are provided - see Section 42.10.6
- The following pair of elements represents an optional attribute which identifies the location of a Metering Device (*this attribute is not certifiable in SE 1.1.1 or earlier and is for future use*):

  - sSiteId is a tsZCL_OctetString structure containing information on the site identifier. This element is paired with au8SiteId (below)
  - au8SiteId is an array containing the site identifier. This element is paired with sSiteId (above)

Note: This identifier is known in the UK as the M-PAN for electricity and MPRN for gas, and in South Africa as as the 'Stand Point'. The field is large enough to accommodate the number of characters typically used in the UK and Europe (16 digits).

- The following pair of elements represents an optional attribute, which indicates the serial number of a Metering Device (*this attribute is not certifiable in SE 1.1.1 or earlier and is for future use*):
  - sMeterSerialNumber is a tsZCL_OctetString structure containing information on the serial number of a Metering Device. This element is paired with au8SiteId (below)
  - au8MeterSerialNumber is an array containing the serial number of a Metering Device. This element is paired with sMeterSerialNumber (above)
- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*(*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):

```
e8EnergyCarrierUnitOfMeasure
u8EnergyCarrierSummationFormatting
u8EnergyCarrierDemandFormatting
e8TemperatureUnitOfMeasure
u8TemperatureFormatting
```

'Historical Consumption' Attribute Set

- i24InstantaneousDemand is an optional attribute containing the current rate of consumption of the metered resource with respect to time. The unit of measure for the relevant resource is as specified in eUnitOfMeasure
  - If this attribute is used, the metering application should update its value on a regular basis, between once every second and once every five seconds. The attribute value can be negative, meaning that the relevant resource is currently being supplied from the premises to the utility company - for example, the case of locally generated electricity from roof-mounted solar panels being supplied to the national grid.
- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*:

```
u24CurrentDayConsumptionDelivered
u24CurrentDayConsumptionReceived
u24PreviousDayConsumptionDelivered
u24PreviousDayConsumptionReceived
utctCurrentPartialProfileIntervalStartTimeDelivered
utctCurrentPartialProfileIntervalStartTimeReceived
u24CurrentPartialProfileIntervalValueDelivered
u24CurrentPartialProfileIntervalValueReceived
```

- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*(*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):

```
u48CurrentDayMaxPressure
u48CurrentDayMinPressure
u48PreviousDayMaxPressure
```

(continues on next page)

(continued from previous page)

```
u48PreviousDayMinPressure
i24CurrentDayMaxDemand
i24PreviousDayMaxDemand
i24CurrentMonthMaxDemand
i24CurrentYearMaxDemand
i24CurrentDayMaxEnergyCarrierDemand
i24PreviousDayMaxEnergyCarrierDemand
i24CurrentMonthMaxEnergyCarrierDemand
i24CurrentMonthMinEnergyCarrierDemand
i24CurrentYearMaxEnergyCarrierDemand
i24CurrentYearMinEnergyCarrierDemand
```

'Load Profile Configuration' Attribute Set

- u8MaxNumberOfPeriodsDelivered is an optional attribute from the Simple Metering 'Load Profile Configuration' attribute set and is fully described in the *ZigBee Smart Energy Profile Specification*.

'Supply Limit' Attribute Set

- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*:

```
u24CurrentDemandDelivered
u24DemandLimit
u8DemandIntegrationPeriod
u8NumberOfDemandSubintervals
```

'Block Information' Attribute Set

- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*(*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):

```
 au48CurrentNoTierBlockSummationDelivered[CLD_SM_ATTR_NO_TIER_BLOCK_CU
RRENT_SUMMATION_DELIVERED_MAX_COUNT]
 au48CurrentTier1BlockSummationDelivered[CLD_SM_ATTR_NUM_OF
_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
 au48CurrentTier2BlockSummationDelivered[CLD_SM_ATTR_NUM_
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
 au48CurrentTier3BlockSummationDelivered[CLD_SM_ATTR_NUM_
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
 au48CurrentTier4BlockSummationDelivered[CLD_SM_ATTR_NUM_
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
 au48CurrentTier5BlockSummationDelivered[CLD_SM_ATTR_NUM_
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
 au48CurrentTier6BlockSummationDelivered[CLD_SM_ATTR_NUM_
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
 au48CurrentTier7BlockSummationDelivered[CLD_SM_ATTR_NUM_
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
 au48CurrentTier8BlockSummationDelivered[CLD_SM_ATTR_NUM_
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
 au48CurrentTier9BlockSummationDelivered[CLD_SM_ATTR_NUM_
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
 au48CurrentTier10BlockSummationDelivered[CLD_SM_ATTR_NUM_
```

(continues on next page)

(continued from previous page)

```
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
  au48CurrentTier11BlockSummationDelivered[CLD_SM_ATTR_NUM_
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
  au48CurrentTier12BlockSummationDelivered[CLD_SM_ATTR_NUM_
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
  au48CurrentTier13BlockSummationDelivered[CLD_SM_ATTR_NUM_
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
  au48CurrentTier14BlockSummationDelivered[CLD_SM_ATTR_NUM_
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
  au48CurrentTier15BlockSummationDelivered[CLD_SM_ATTR_NUM_
OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]
```

Parent topic:*Simple Metering Cluster*

Attribute Settings The Simple Metering cluster contains both mandatory and optional attributes (see Section 42.2). The cluster structure is shown below with only the mandatory attributes (which are enabled by default):

```
typedef struct PACK
{
  zuint48                    u48CurrentSummationDelivered;
  zbmap8                     u8MeterStatus;
  teSE_UnitOfMeasure         eUnitOfMeasure;
  zbmap8                     u8SummationFormatting;
  teSE_MeteringDeviceType    eMeteringDeviceType;
} tsSE_SimpleMetering;
```

The mandatory attribute settings are outlined below.

eMeteringDeviceType The element eMeteringDeviceType of the structure tsSE_SimpleMetering indicates the type of Metering Device in terms of the resource type which it measures: electricity, gas, water, heat, cooling or pressure. This attribute belongs to the cluster's Formatting attribute set.

Enumerated values are provided for the full range of possible metering devices - for example, E_CLD_SM_MDT_GAS for a gas meter. Enumerated values are also provided for devices that mirror a Metering Device - for example, E_CLD_SM_MDT_GAS_MIRRORED for the mirroring device of a gas meter. All of these enumerations are defined in the structure teCLD_SM_MeteringDeviceType, detailed in Section 42.10.6.

u8MeterStatus The element u8MeterStatus of the structure tsSE_SimpleMetering indicates the current status of the device by means of an 8-bit value. This attribute has its own attribute set, Meter Status.

The status value is a bitmap with the bit representations indicated in the table below:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Service						

Disconnect

Open *

|Leak Detect|Power Quality|Power

Failure

|Tamper Detect|Battery Low|Check Meter|

* Set to '1' when service to this site has been disconnected

A bit is set (to '1') to indicate the corresponding error or warning.

A number of macros are defined to reflect the above bit settings - for example, E_CLD_SM_METER_STATUS_POWER_FAILURE_BIT contains the state of the Power Failure bit (Bit 3). There are also macros for masking off the appropriate bit - these macros are detailed in Section 42.10.2.

eUnitOfMeasure The element eUnitOfMeasure of the structure tsSE_SimpleMetering indicates the unit of measure in which the relevant resource is metered, e.g. kiloWatt-hour for electricity. This attribute belongs to the cluster's Formatting attribute set.

Enumerated values are provided for the possible units of measure - for example, E_CLD_SM_UOM_CUBIC_METER for cubic metre (of gas or water). This example will also configure measurements to be expressed in binary/hex. However, enumerated values are also provided to configure measurements to be expressed in binary coded decimal - for example, E_CLD_SM_UOM_CUBIC_METER_BCD configures measurements in cubic metres and expressed in binary coded decimal. All of these enumerations are defined in the structure teCLD_SM_UnitOfMeasure, detailed in Section 42.10.3.

u8SummationFormatting The element u8SummationFormatting of the structure tsSE_SimpleMetering is an 8-bit value indicating the position of the decimal point in the metered value (see *u48CurrentSummationDelivered*). This attribute belongs to the cluster's Formatting attribute set.

This value contains bit fields, as follows:

- **Bits 2-0:** 3-bit value indicating number of digits to right of point
- **Bits 6-3:** 3-bit value indicating number of digits to left of point
- **Bit 7:** Setting this bit (to '1') suppresses leading zeros

A number of macros are defined to accommodate the above format information - these macros are detailed in Section 42.10.4.

u48CurrentSummationDelivered The element u48CurrentSummationDelivered of the structure tsSE_SimpleMetering is a 48-bit value representing the total quantity consumed, so far, of the metered resource (e.g. electrical energy). This attribute belongs to the Reading Information attribute set.

The attribute value is interpreted with the aid of the elements eUnitOfMeasure and u8SummationFormatting, which indicate the unit of measure and the position of the decimal point respectively.

Parent topic:*Simple Metering Cluster*

Remotely Reading Simple Metering Attributes Dedicated functions are provided for remotely reading the Simple Metering attributes:

1. The application must first call **eSE_ReadMeterAttributes()** to submit a 'read attributes' request to the relevant remote endpoint. The resulting read process is as described for **eZCL_SendReadAttributesRequest()** in Section 2.3.2.

2. On receiving the 'read attributes' response, the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE is generated, which causes the callback function for the local endpoint to be invoked. This callback function should include a call to **eSE_HandleReadAttributesResponse()** which checks whether all the Simple Metering attributes are included in the response. If the response is not complete, the function re-sends 'read attributes' requests until all attribute values are obtained.

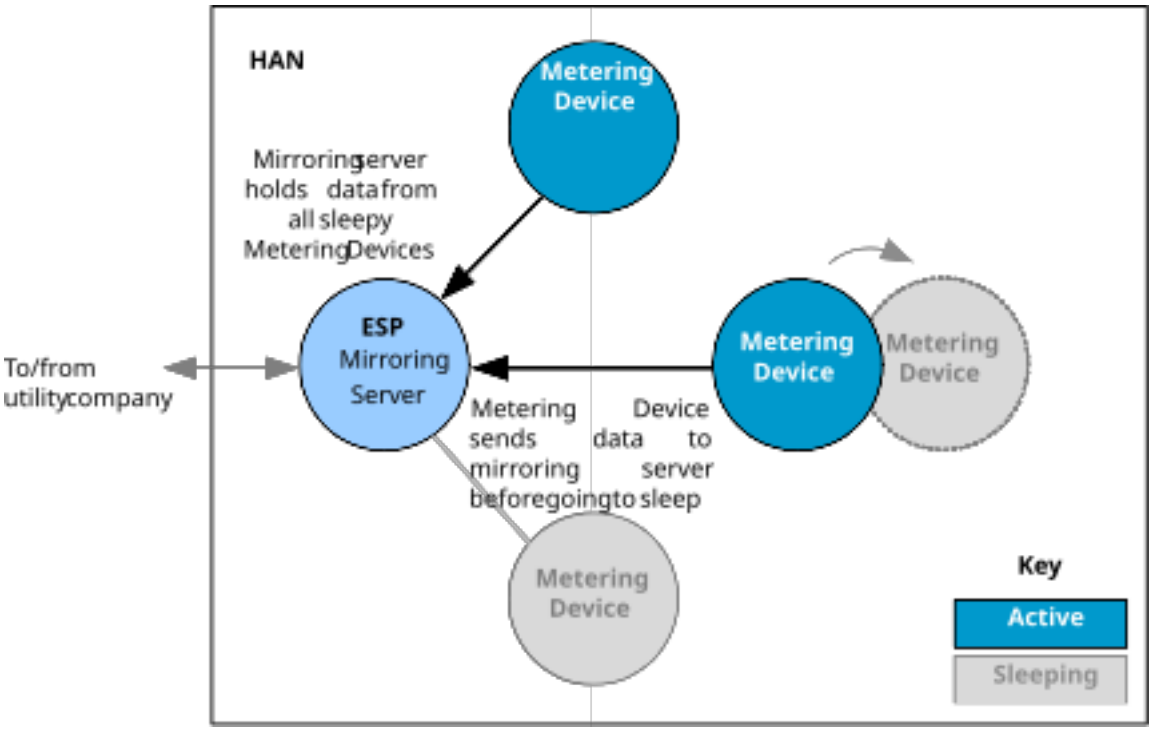
Note that read access to cluster attributes must be explicitly enabled at compile-time as described in Section 1.3.

Parent topic:*Simple Metering Cluster*

Mirroring Metering Data 'Mirroring' is a facility that stores and provides access to metering data which originates from Metering Devices that sleep. A Metering Device cannot be accessed during periods of sleep and therefore its data cannot normally be read at these times. Mirroring involves holding the data from sleepy Metering Devices centrally on a server, allowing access to the data at all times.

Normally, the ESP (Co-ordinator) acts as the mirroring server. One or more sleepy Metering Devices (End Devices) can mirror their data on this server. A Metering Device must send its latest data to the mirroring server immediately before entering sleep mode. This is illustrated in the figure below.

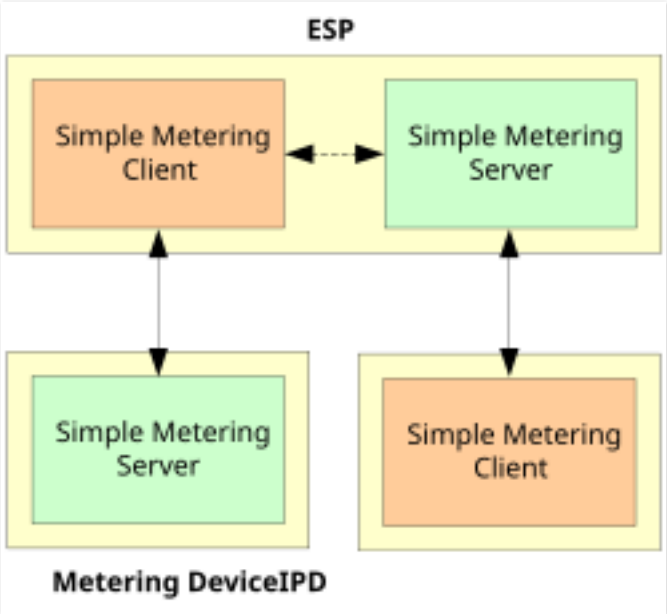
Mirroring of Metering Data



Every mirror (one for each Metering Device) on the mirroring server has its own endpoint. The maximum number of mirror endpoints is defined at compile-time (see Section 42.12). Note that these endpoints are in addition to the main endpoint for the ESP (registered using **eSE_RegisterEspMeterEndPoint()** or **eSE_RegisterEspEndPoint()**).

Mirroring versions of the Simple Metering cluster server and/or client are implemented on the mirror endpoints. This is illustrated in the figure below, where the ESP, as the mirroring server, incorporates both the Simple Metering cluster server and client, the Metering device incorporates a cluster server and the IPD incorporates a cluster client.

Simple Metering Cluster in Mirroring



The ESP device structure tsSE_EspMeterDevice contains a section on mirroring support which includes an array of tsSE_Mirror structures (see Section 42.11.2). This array contains one element/structure per mirror endpoint, with the first mirror endpoint occupying array element 0 and the array size corresponding to the maximum number of mirror endpoints allowed on the mirroring server. The information stored in an array element includes the IEEE address of the Metering Device to which the mirror endpoint has been allocated.

Configuring Mirroring on ESP The ESP normally acts as the mirroring server, containing a unique mirror endpoint for each (mirrored) Metering Device. Configuration of the mirroring server is carried out both within the application that runs on the device and as compile-time options - refer to Section 42.12 for the relevant compile-time options.

On the ESP, mirroring can be enabled in the application code when the device is registered using the function **eSE_RegisterEspMeterEndPoint()** or **eSE_RegisterEspEndPoint()**. These functions require specification of the first endpoint that is to be used for mirroring. Starting at this endpoint, consecutive endpoints to be used for mirrors are reserved, up to the maximum number of mirrors defined by the compile-time option CLD_SM_NUMBER_OF_MIRRORS. For example, if 5 is specified as the first mirror endpoint and up to 4 mirrors can be used then endpoints 5, 6, 7 and 8 are reserved for mirrors. Note that mirroring is disabled by setting the start endpoint to 0.

Note: The endpoints reserved for mirroring must also be included in the configuration diagram in the ZPS Configuration Editor. However, they must not be enabled since they are enabled when mirrors are created on them.

The tsSE_Mirror structures in the ESP device structure tsSE_EspMeterDevice contain the IEEE addresses of the Metering Devices being mirrored on the ESP (these IEEE addresses are automatically initialised to zero). The ESP application must save an array of these IEEE addresses to non-volatile memory using the NVM module - this will allow the mirrored Metering Devices to be identified by the mirroring server following a reset of the ESP.

The ESP must allocate mirror endpoints to Metering Devices in response to requests from the Metering Devices (refer to Section 42.5.2 for details of requesting a mirror), as described below:

1. On receiving a mirror request on the ESP, the ZCL automatically allocates the next available mirror endpoint to the Metering Device (the IEEE address of the Metering Device is automatically written to the tsSE_Mirror structure which corresponds to the allocated mirror endpoint).

2. The event E_CLD_SM_CLIENT_RECEIVED_COMMAND containing the command E_CLD_SM_REQUEST_MIRROR is then generated on the ESP, causing the callback function on the ESP to be invoked.
3. The callback function must check whether all mirror endpoints have now been exhausted, in order to update the relevant status on the ESP. To do this, the function **eSM_GetFreeMirrorEndPoint()** must be called to obtain the number of the next free mirror endpoint. If the value 0xFFFF is returned, this means that no more mirror endpoints are available (for subsequent requests) and the attribute u8PhysicalEnvironment of the Basic cluster must be set to zero (to indicate to other Metering Devices that no more mirrors are available on the ESP). This step is illustrated in the code fragment below.

```
eSM_GetFreeMirrorEndPoint (&u16FoundEP);
if (u16FoundEP == 0xFFFF)
{
        psSE_EspMeterDevice->sBasicCluster.u8PhysicalEnvironment = 0x00;
}
else
{
        psSE_EspMeterDevice->sBasicCluster.u8PhysicalEnvironment = 0x01;
}
```

4. The callback function must copy the IEEE addresses from the tsSE_Mirror structures (which are automatically kept up-to-date) to the application's array of IEEE addresses for mirrored devices, and this array should be re-saved in non-volatile memory using the NVM module. This step is illustrated below in the code fragment under "*Writing and Preserving Array of IEEE Addresses*".
5. A response is automatically sent to the requesting Metering Device, where this response contains the number of the assigned endpoint.

The ESP is then ready to receive metering data from the remote Metering Device, as described in Section 42.5.3.

Writing and Preserving Array of IEEE Addresses The ESP application must maintain an array of the IEEE addresses of the mirrored Metering Devices and keep a copy of this array in NVM. The array can be updated from the tsSE_Mirror structures for the mirror endpoints and saved to NVM as illustrated in the code fragment below:

```
case E_UPDATE_EVENT_REQUEST_MIRROR:
case E_UPDATE_EVENT_REMOVE_MIRROR:
{
  uint8 u8LoopCntr;
  for (u8LoopCntr =0; u8LoopCntr < CLD_SM_NUMBER_OF_MIRRORS; u8LoopCntr++)
     {
       sMirrorState.u64ExtAddr[u8LoopCntr] =
                 sMeter.sSE_Mirrors[u8LoopCntr].u64SourceAddress;
     }
     sMirrorState.bNetworkUp = TRUE;
    NvSaveOnIdle(&sMirrorState, TRUE);
}
break;
```

Recreating Mirrors Following an ESP Reset If the ESP is reset, the mirrors that have been created on the device are lost. However, if the IEEE addresses (of the mirrored Metering Devices) associated with the mirror endpoints have been preserved in NVM, this data can be read by the ESP application following the reset and the mirrors recreated. Given the relevant endpoint number and IEEE address, a mirror can be recreated using the function **eSM_CreateMirror()**.

Note: A matching function **eSM_RemoveMirror()**also exists to allow the application to remove a mirror.

Parent topic: Mirroring Metering Data

Configuring Mirroring on Metering Devices Configuration of a Metering Device for mirroring is carried out both within the application that runs on the device and as a compile-time option - refer to Section42.12 for the relevant compile-time options.

It is the responsibility of the Metering Device to request a mirror on the ESP, but first it must establish whether the ESP is accepting mirror requests. To do this, the application should use the function **eZCL_SendReadAttributesRequest()** to obtain the value of the u8PhysicalEnvironment attribute of the Basic cluster on the ESP - if this value is non-zero then the ESP is open to receiving mirror requests.

Provided that the ESP is accepting mirror requests, a Metering Device application can request a mirror using the function **eSM_ServerRequestMirrorCommand()**. This function sends a mirror request to the ESP with the aim of being allocated a mirror endpoint. The handling of this request on the ESP is described in Section 42.5.1.

The Metering Device application must then wait for a response from the ESP. This response is indicated by the event E_CLD_SM_SERVER_RECEIVED_COMMAND containing the command E_CLD_SM_REQUEST_MIRROR_RESPONSE, causing the callback function for the receiving endpoint to be invoked.

If the request has resulted in the successful allocation of a mirror endpoint on the ESP, the tsSM_RequestMirrorResponseCommand structure (see Section 42.11.6) in this event will contain the allocated endpoint number. In this case:

- The callback function should write the allocated endpoint number and mirroring server (ESP) IEEE address to non-volatile memory for persistent data storage using the NVM module.
- The Metering Device application can now send metering data for storage on the ESP whenever required, as described in Section 42.5.3.

Note: If the Metering Device subsequently requests another mirror on the same ESP, the same mirror endpoint number will be returned - a Metering Device cannot have more than one mirror on the same ESP.

If the request did not result in an allocated mirror endpoint on the ESP, the endpoint number returned in the above structure will be 0xFFFF and no action needs to be taken by the callback function.

Parent topic: Mirroring Metering Data

Mirroring Data Once a mirror for a Metering Device has been set up, as described in Section 42.5.1 and Section 42.5.2, the mirror can be populated and refreshed with data in two ways:

- The ESP application can submit a 'read attributes' request to the Metering Device (when it is not asleep), as described in Section 2.3.2.
- The Metering Device can send metering data as unsolicited attribute reports to the mirror at any time (for example, before entering sleep mode). This method is described further below.

The Metering Device application sends unsolicited attribute reports for the Simple Metering cluster to the mirror using the function **eZCL_ReportAllAttributes()**, described in Section 5.2.

On receiving this data, the event E_ZCL_CBET_ATTRIBUTE_REPORT_MIRROR is generated on the ESP, causing the callback function on the ESP to be invoked. The callback function must then check that the data has come from a valid source (a Metering Device which has a mirror on the

ESP) by calling the function **eSM_IsMirrorSourceAddressValid()**. According to the outcome of this check, the function updates the event status:

```
sZCL_CallBackEvent.uMessage.sReportAttributeMirror.eStatus
```

- If eStatus is set to E_ZCL_ATTR_REPORT_OK, the reported attribute values (metering data) are automatically stored on the relevant mirror endpoint and an E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTE event is generated for each attribute reported.
- If eStatus is set to anything else, a ZCL default response is automatically sent back to the reporting device to indicate that mirroring is not authorised for this device (E_ZCL_CMDS_NOT_AUTHORIZED).

Maintaining the Mirrored eMeteringDeviceType Attribute When a mirror is created on the ESP, the Simple Metering cluster attribute eMeteringDeviceType in the mirror will be set to the appropriate value for the Metering Device to be mirrored (e.g. E_CLD_SM_MDT_GAS). However, in order to distinguish the mirror cluster on the ESP from the original cluster on the Metering Device, the ESP application must replace this value in the mirror with the equivalent '_MIRRORED' value (e.g. E_CLD_SM_MDT_GAS_MIRRORED). In fact, this replacement must be performed every time the ESP receives a new set of attribute values from the Metering Device (by either of the two methods described above), since this attribute value in the mirror will be over-written each time and must subsequently be corrected.

Parent topic: Mirroring Metering Data

Reading Mirrored Data A ZigBee device such as an IPD may need to obtain data from a mirror on the ESP, particularly when the mirrored Metering Device is sleeping. The data is requested by means of the standard 'read attributes' method, described in Section 2.3.2 - that is, by calling the ZCL function **eZCL_SendReadAttributesRequest()** on the requesting device.

If an attempt is made to read an attribute that currently has no value in the mirror, the resulting E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE event will contain the attribute status E_ZCL_CMDS_UNSUPPORTED_ATTRIBUTE.

Parent topic: Mirroring Metering Data

Removing a Mirror The removal of a mirror on the ESP is initiated by the application on the corresponding Metering Device using the function **eSM_ServerRemoveMirrorCommand()**. This function sends a 'remove mirror' request to the relevant mirror endpoint on the ESP.

Note: A mirror can be removed from an endpoint on the ESP but the endpoint will remain reserved for mirroring - it may later be re-assigned to another mirror.

On receiving this request, the ESP processes the request as follows:

1. The ZCL first verifies the source address of the request to ensure that it has come from the Metering Device which corresponds to the mirror to be removed. If the source address is not valid then a ZCL default response is automatically sent to the requesting Metering Device to indicate that the request was not authorised (E_ZCL_CMDS_NOT_AUTHORIZED) - otherwise, the ESP continues to process the request as described in the steps below.
2. The ZCL then removes the mirror from the specified endpoint, thus freeing the endpoint for future use by another mirror.
3. The event E_CLD_SM_CLIENT_RECEIVED_COMMAND containing the command E_CLD_SM_REMOVE_MIRROR is generated on the ESP, causing the callback function on the ESP to be invoked.

4. The callback function must set the u8PhysicalEnvironment attribute of the Basic cluster to 0x01 in order to indicate that the ESP has the capacity to accept mirror requests (since the removal of the mirror leaves at least one mirror endpoint free).
5. The callback function must copy the IEEE addresses from the tsSE_Mirror structures (which are automatically kept up-to-date) to the application's array of IEEE addresses for mirrored devices, and this array should be re-saved in non-volatile memory using the NVM module. This step is illustrated in the code fragment under "Writing and Preserving Array of IEEE Addresses" on page 1011.
6. A response is automatically sent to the requesting Metering Device to confirm the mirror removal.

The response (reporting successful mirror removal) results in the generation of the event E_CLD_SM_SERVER_RECEIVED_COMMAND containing the command E_CLD_SM_MIRROR_REMOVED on the Metering Device.

Note: The function **eSM_RemoveMirror()** is also provided, which allows the ESP application to directly remove a mirror.

Parent topic: Mirroring Metering Data

Parent topic:*Simple Metering Cluster*

Consumption Data Archive ('Get Profile') Devices that support the Simple Metering cluster can maintain and exchange historical consumption (profiling) data using the 'Get Profile' feature. A consumption data archive, which is distinct from the data of the Simple Metering cluster attributes, is maintained in a circular buffer on the cluster server. A cluster client can make a 'Get Profile' request to the server to obtain data from this archive. Normally, the cluster server is implemented on a Metering Device and the cluster client is implemented on an IPD. Typically, the IPD requests a consumption history from the Metering Device in order to display this information to the consumer.

The consumption data in the archive corresponds to a series of consecutive time intervals with their corresponding consumption values. Thus, the archive consists of the last few consumption measurements - it is the responsibility of the application running on the server device to update the archive (see Section 42.6.1).

If the 'Get Profile' feature is required, it must be enabled in the compile-time options as described in Section 42.12. These options include the maximum number of consumption intervals that can be archived on the server (and therefore requested).

Updating Consumption Data on Server The consumption archive is held on the Smart Metering cluster server in a circular buffer operating on a FIFO basis. This buffer provides storage space for a sequence of entries containing consumption data for consecutive time intervals, where each buffer entry is a structure of the type tsSEGetProfile consisting of:

- End-time of consumption interval (as UTC time)
- Units delivered to the customer
- Units received from the customer (when customer sells units to utility company)

The maximum number of entries that can be stored in the buffer is determined at compile-time (see Section 42.12). When a new entry is added to a full buffer, this entry replaces the oldest entry currently in the buffer.

The application must keep the buffer up-to-date by adding a new entry using the function **eSM_ServerUpdateConsumption()**. Before this function is called, the relevant consumption data must be updated in one or both of the following Simple Metering cluster attributes:

```
u24CurrentPartialProfileIntervalValueDelivered
```

- Contains the number of units delivered to the customer over the last interval

```
u24CurrentPartialProfileIntervalValueReceived
```

- Contains the number of units received from the customer over the last interval

An attribute only needs to be updated if the corresponding consumption has been implemented (for example, the utility company often only delivers units to the customer and does not receive any from the customer).

eSM_ServerUpdateConsumption() takes the current time as an input and then adds an entry containing the consumption data (in the above attributes) to the buffer, where the supplied current time becomes the end-time in the entry (thus, the duration of the consumption intervals is dictated by the frequency at which this function is called - see below).

Note: The current time can be obtained by the application using the function **u32ZCL_GetUTCTime()**, described in Section 18.7.

eSM_ServerUpdateConsumption() must be called periodically by the application. The period must match the value to which the Simple Metering eProfileIntervalPeriod attribute has been set (see Section 42.2). Standard periods, ranging from 2.5 minutes to one day, are provided as a set of enumerations (see Section 42.10.10).

Parent topic: Consumption Data Archive ('Get Profile')

Sending and Handling a 'Get Profile' Request The application on a device which supports the Simple Metering cluster as a client, such as an IPD, can send a 'Get Profile' request to the cluster server by calling the function **eSM_ClientGetProfileCommand()**. This function allows consumption data to be requested from the archive for one or more intervals.

The inputs for this function include:

- A value indicating whether the units delivered or units received (by the utility company) are being requested (see Section 42.6.1)
- An end-time (as a UTC time) - the most recent consumption data will be reported which has an end-time equal to or earlier than this end-time (a specified end-time of zero will result in the most recent consumption data)
- The number of consumption intervals to report (this number will be reported only if data for sufficient intervals is available) - the end-time rule, specified above, will be applied to all the reported intervals

On receiving the request, the event E_CLD_SM_SERVER_RECEIVED_COMMAND containing the command E_CLD_SM_GET_PROFILE is generated on the server, causing the callback function on the device to be invoked (for a Metering Device, this is the callback function registered through **eSE_RegisterEspMeterEndPoint()** or **eSE_RegisterMeterEndPoint()**). The callback function only needs to be concerned with this event if the archive data needs to be modified before the ZCL automatically sends the requested data in a 'Get Profile' response. The response indicates the number of consumption intervals reported and contains the consumption data for these intervals, as well as the end-time of the most recent interval reported.

On receiving the response, the event E_CLD_SM_CLIENT_RECEIVED_COMMAND containing the command E_CLD_SM_GET_PROFILE_RESPONSE is generated on the requesting client, causing the callback function on the device to be invoked (for an IPD, this is the callback function registered through **eSE_RegisterIPDEndPoint()**). The callback function should extract the requested data from the event using the function **u32SM_GetReceivedProfileData()** in order to process or store the data. This function should be called for each consumption interval reported in the event - the code fragment below illustrates repeated calls to the function until all the reported data has been obtained:

```
for (i =0 ;i < sGetProfileResponseCommand.u8NumberOfPeriodsDelivered; i
++)
{
   //Read data from event
X(i)= u32SM_GetReceivedProfileData(tsSM_
↪GetProfileResponseCommand *psSMGetProfileResponseCommand)
}
```

Alternatively, the function can be called repeatedly until it returns 0xFFFFFFFF, which indicates that there is no more data to be extracted from the event.

Parent topic: Consumption Data Archive ('Get Profile')

Parent topic:*Simple Metering Cluster*

Simple Metering Events The Simple Metering cluster has its own events that are handled through the callback mechanism described in *Chapter 3*. If a device uses the Simple Metering cluster then Simple Metering event handling must be included in the callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function (for example, through **eSE_RegisterMeterEndPoint()** for a standalone Metering Device). The relevant callback function will then be invoked when a Simple Metering event occurs.

For a Simple Metering event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to a tsSM_CallBackMessage structure which contains the Simple Metering parameters:

```
typedef struct
{
  teSM_CallBackEventType eEventType;
  uint8 u8CommandId;
    union
    {
      tsSM_GetProfileResponseCommand      sGetProfileResponseCommand;
      tsSM_RequestFastPollResponseCommand     sRequestFastPollResponseCommand;
      tsSM_GetProfileRequestCommand      sGetProfileCommand;
      tsSM_RequestMirrorResponseCommand     sRequestMirrorResponseCommand;
      tsSM_MirrorRemovedResponseCommand     sMirrorRemovedResponseCommand;
      tsSM_RequestFastPollCommand       sRequestFastPollCommand;
      tsSM_Error              sError;
    }uMessage;
}tsSM_CallBackMessage;
```

Information on the elements of the above structure is provided below.

Event Types The eEventType field of the tsSM_CallBackMessage structure specifies the type of Simple Metering event that has been generated. These event types are enumerated in the teSM_CallBackEventType structure (see Section 42.10.7) and are listed in the table below.

Event Type Enumeration	Description
E_CLD_SM_CLIENT_RECEIVED_C(	Generated when a command has been received on a cluster client
E_CLD_SM_SERVER_RECEIVED_C	Generated when a command has been received on the cluster server
E_CLD_SM_FAST_POLLING_TIME	Generated on the cluster server at the end of a fast polling episode (*for future use*)

The possible command types for the above event types are listed in Section 42.7.2.

Parent topic: Simple Metering Events

Command Types For each event type listed in Section 42.7.1, one of a number of command types could have been received. The relevant command type is specified through the u8CommandId field of the tsSM_CallBackMessage structure. The possible command types for each event type are detailed below.

E_CLD_SM_CLIENT_RECEIVED_COMMAND The E_CLD_SM_CLIENT_RECEIVED_COMMAND event is generated when a command has been received on a cluster client. The possible command types for this event type are listed in the table below, which gives the enumerations and the associated uMessage union elements in the tsSM_CallBackMessage structure:

u8CommandId Enumeration	uMessage Union Element
E_CLD_SM_GET_PROFILE_RESPONSE	sGetProfileResponseCommand
E_CLD_SM_REQUEST_MIRROR	sRequestMirrorAdd
E_CLD_SM_REMOVE_MIRROR	sRequestMirrorRemove
E_CLD_SM_REQUEST_FAST_POLL_MODE_	

RESPONSE

|sRequestFastPollResponseCommand

(*for future use*)

| |E_CLD_SM_CLIENT_ERROR|sError|

The above command enumerations are fully described in Section 42.10.8.

E_CLD_SM_SERVER_RECEIVED_COMMAND The E_CLD_SM_SERVER_RECEIVED_COMMAND event is generated when a command has been received on the cluster server. The possible command types for this event type are listed in the table below, which gives the enumerations and the associated uMessage union elements in the tsSM_CallBackMessage structure:

u8CommandId Enumeration	uMessage Union Element
E_CLD_SM_GET_PROFILE	sGetProfileCommand
E_CLD_SM_REQUEST_MIRROR_RESPONSE	sRequestMirrorResponseCommand
E_CLD_SM_MIRROR_REMOVED	sMirrorRemovedResponseCommand
E_CLD_SM_REQUEST_FAST_POLL_MODE	sRequestFastPollCommand (*for future use*)
E_CLD_SM_SERVER_ERROR	sError

The above command enumerations are fully described in Section 42.10.9.

E_CLD_SM_FAST_POLLING_TIMER_EXPIRED The E_CLD_SM_FAST_POLLING_TIMER_EXPIRED event is generated on the cluster server at the end of a fast polling episode. It has no associated data structure. *Fast polling is not certifiable in SE 1.1.1 or earlier and this event is reserved for future use*.

Parent topic: Simple Metering Events

Parent topic:*Simple Metering Cluster*

Functions The following Simple Metering cluster functions are provided:

1. eSE_SMCreate
2. eSE_ReadMeterAttributes
3. eSE_HandleReadMeterAttributesResponse
4. eSM_ServerRequestMirrorCommand
5. eSM_ServerRemoveMirrorCommand
6. eSM_CreateMirror
7. eSM_RemoveMirror
8. eSM_GetFreeMirrorEndPoint
9. eSM_IsMirrorSourceAddressValid
10. eSM_ServerUpdateConsumption
11. eSM_ClientGetProfileCommand
12. u32SM_GetReceivedProfileData

eSE_SMCreate

```
teZCL_Status eSE_SMCreate(
    uint8 u8Endpoint,
    bool_t bIsServer,
    uint8 *pu8AttributeControlBits,
    tsZCL_ClusterInstance *psClusterInstance,
    tsZCL_ClusterDefinition *psClusterDefinition,
    tsSM_CustomStruct *psCustomDataStruct,
    void *pvEndPointSharedStructPtr);
```

Description This function creates an instance of the Simple Metering cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Simple Metering cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to *Appendix D.*

Note: This function must not be called for an endpoint on which a standard ZigBee device (for example, an IPD) is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in the *ZigBee Devices User Guide (JNUG3131).*

When used, this function must be the first Simple Metering cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate be the total number of attributes supported by the Simple Metering cluster, which can be obtained by using the macro SM_NUM_OF_ATTRIBUTES.

The array declaration should be as follows:

```
uint8 au8AppSMClusterAttributeControlBits[SM_NUM_OF_ATTRIBUTES];
```

The function initializes the array elements to zero.

Parameters

- *u8Endpoint*: Number of local endpoint on which the cluster instance is to be created, in the range 1 to 240.
- *bIsServer* : Type of cluster instance (server or client) to be created:
- : TRUE - server
- : FALSE - client
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Simple Metering cluster. This parameter can refer to a pre-filled structure called sCLD_SimpleMetering which is provided in the **SimpleMetering.h** file.
- *psCustomDataStructure*: Pointer to structure which contains custom data for the Simple Metering cluster. This structure is used for internal data storage and also contains data relating to a received command/message. No knowledge of the fields of this structure is required.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_SimpleMetering which defines the attributes of Simple Metering cluster. The function will initialize the attributes with default values.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eSE_ReadMeterAttributes

```
teZCL_Status eSE_ReadMeterAttributes(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used to send a 'read attributes' request to the Simple Metering cluster on a remote endpoint. The function requests all Simple Metering attributes to be read - alternatively, the function **eZCL_SendReadAttributesRequest()** can be used if only specific attributes are required. Note that read access to cluster attributes on the remote node (server) and local node (client) must be enabled at compile-time, as described in Section 1.3.

You must specify the endpoint on the local node from which the request is to be sent. This is also used to identify the instance of the local shared device structure which holds the relevant attributes. The obtained attribute values are written to this shared structure by the function.

You must also specify the address of the destination node and the destination endpoint number. It is possible to use this function to send a request to bound endpoints or to a group of endpoints on remote nodes - in the latter case, a group address must be specified. Note that when sending requests to multiple endpoints through a single call to this function, multiple responses will subsequently be received from the remote endpoints.

You are also required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Following the first response to this function call, your application should call the function **eSE_HandleReadAttributesResponse()** to ensure that all the Simple Metering attributes are received from the remote endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent. Note that this parameter is ignored when sending to address types E_ZCL_AM_BOUND and E_ZCL_AM_GROUP
- *psDestinationAddress*: Pointer to a structure containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_CLUSTER_ID_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTE_WO
- E_ZCL_ERR_ATTRIBUTES_ACCESS
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_PARAMETER_RANGE

Parent topic: Functions

eSE_HandleReadMeterAttributesResponse

```
teSE_Status eSE_HandleReadAttributesResponse(
    tsZCL_CallBackEvent *psEvent,
    uint8 *puTransactionSequenceNumber);
```

Description This function should be called after **eSE_ReadMeterAttributes()**. The function examines the response to a 'read attributes' request for the Simple Metering cluster and determines whether the response is complete - that is, whether it contains all the Simple Metering attributes (the response may be incomplete if the returned data is too large to fit into a single APDU). If the response is not complete, the function will re-send 'read attributes' requests until all attribute values have been obtained. Any further attribute values obtained are written to the local shared device structure containing the attributes.

This function call should normally be included in the user-defined callback function that is invoked when the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE is generated. This is the callback function which is specified when the (requesting) endpoint is registered using the appropriate endpoint registration function. The callback function must pass the generated event into **eSE_HandleReadAttributesResponse()**.

You are also required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request/response.

Parameters

- *psEvent*: Pointer to the generated event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request/response

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_CLUSTER_ID_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTE_WO
- E_ZCL_ERR_ATTRIBUTES_ACCESS
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_PARAMETER_RANGE

Parent topic: Functions

eSM_ServerRequestMirrorCommand

```
teZCL_Status eSM_ServerRequestMirrorCommand(
    uint8 u8SourceEndpoint,
    uint8 u8DestinationEndpoint,
    tsZCL_Address *psDestinationAddress);
```

Description This function can be used by a Metering Device to request a mirror on the ESP, for the central storage of its metering data. A mirror is useful for a Metering Device which sleeps, in order to allow access to its metering data while the device is sleeping.

The function sends an 'Add Mirror' request to the ESP. The address of the ESP device must be specified as well as the endpoint that will receive and process the request - this is the main endpoint on which the ESP is registered on the Co-ordinator. If successful, the request will result in the allocation of a mirror endpoint (on the ESP) to the Metering Device.

Note: Before using this function to send an 'Add Mirror' request, the Metering Device application should check whether the ESP is currently accepting these requests by calling the function **eZCL_SendReadAttributesRequest()** to obtain the value of the u8PhysicalEnvironment attribute of the Basic cluster on the ESP. This attribute value will be non-zero if 'Add Mirror' requests are being accepted.

eSM_ServerRequestMirrorCommand() is a non-blocking function and so returns immediately after the request has been sent. The application must then wait for a response, indicated by the event E_CLD_SM_SERVER_RECEIVED_COMMAND containing the command E_CLD_SM_REQUEST_MIRROR_RESPONSE. If a mirror was successfully created, the number of the allocated mirror endpoint on the ESP is included in the event.

Mirroring and mirror set-up are fully described in Section 42.5.

Parameters

- *u8SourceEndpoint*: Number of local endpoint through which request is sent
- *u8DestinationEndpoint*: Number of ESP endpoint to which request is sent (main endpoint of ESP)
- *psDestinationAddress*: Pointer to a structure containing the address of the ESP device (to which the request is sent)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_CLUSTER_ID_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_EP_RANGE

Parent topic: Functions

eSM_ServerRemoveMirrorCommand

```
teZCL_Status eSM_ServerRemoveMirrorCommand(
    uint8 u8SourceEndpoint,
    uint8 u8DestinationEndpoint,
    tsZCL_Address *psDestinationAddress);
```

Description This function can be used on a Metering Device to request the removal of the corresponding mirror on the ESP. The function should only be used to remove a mirror that has been previously set up by the Metering Device application using the function **eSM_ServerRequestMirrorCommand()**.

The function sends a 'Remove Mirror' request to the ESP. The address of the ESP must be specified as well as the endpoint number of the mirror to be removed.

This is a non-blocking function and so returns immediately after the request has been sent. The application must then wait for a response.

- If the request was successful, a response will be received from the ESP resulting in the generation of the event E_CLD_SM_SERVER_RECEIVED_COMMAND containing the command E_CLD_SM_MIRROR_REMOVED
- If the request was unsuccessful, a ZCL default response will be received from the ESP to indicate that the request was not authorised (E_ZCL_CMDS_NOT_AUTHORIZED)

Mirror removal is fully described in Section 42.5.5.

Parameters

- *u8SourceEndpoint*: Number of local endpoint through which request is sent
- *u8DestinationEndpoint*: Number of ESP endpoint which contains the mirror to be removed
- *psDestinationAddress*: Pointer to a structure containing the address of the ESP device (to which the request is sent)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_CLUSTER_ID_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_EP_RANGE

Parent topic: Functions

eSM_CreateMirror

```
teSM_Status eSM_CreateMirror(
    uint8 u8MirrorEndpoint,
    uint64 u64RemoteIeeeAddress);
```

Description This function can be used on the mirroring server (ESP) to create a mirror with the specified endpoint number for the Metering Device with the specified IEEE address. The endpoint number must be within the valid range for mirror endpoints on the ESP.

An error will be returned if there is no free mirror endpoint on which to create a mirror.

The function is normally used by an ESP application following a device reset, in order to recreate mirrors that were lost during the reset. This recovery assumes that the relevant IEEE addresses (for Metering Devices) associated with the mirror endpoints can be retrieved from non-volatile memory, where they were saved before the reset.

Parameters

- *u8MirrorEndpoint*: Number of endpoint on which mirror will be created (must be within valid range for mirror endpoints)
- *u64RemoteIeeeAddress*: IEEE address of Metering Device to be mirrored

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_RANGE
- E_CLD_SM_STATUS_EP_NOT_AVAILABLE

Parent topic: Functions

eSM_RemoveMirror

```
teSM_Status eSM_RemoveMirror(
    uint8 u8MirrorEndpoint,
    uint64 u64RemoteIeeeAddress);
```

Description This function can be used on the mirroring server (ESP) to remove the mirror with the specified endpoint number for the Metering Device with the specified IEEE address. The endpoint will then become free to be re-allocated for another mirror.

An error will be returned if the specified mirror endpoint cannot be found.

Parameters

- *u8MirrorEndpoint*: Number of endpoint which hosts mirror to be removed (must be within valid range for mirror endpoints)
- *u64RemoteIeeeAddress*: : IEEE address of mirrored Metering Device

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_RANGE
- E_CLD_SM_STATUS_EP_NOT_AVAILABLE
- E_ZCL_FAIL

Parent topic: Functions

eSM_GetFreeMirrorEndPoint

```
teZCL_Status eSM_GetFreeMirrorEndPoint(
    uint16 *pu16FreeEP);
```

Description This function can be used on the mirroring server (ESP) to obtain the number of the next available mirror endpoint. If there are no free mirror endpoints, the function sets the returned endpoint number to 0xFFFF.

The function is normally used in the ESP callback function to check the availability of mirror endpoints before updating the `u8PhysicalEnvironment` attribute of the Basic cluster (this attribute is set to zero if no more mirror endpoints are available).

Use of this function is described in Section 42.5.1.

Parameters

- *pu16FreeEP*: Pointer to location to receive next free endpoint number

Returns

- E_ZCL_SUCCESS

Parent topic: Functions

eSM_IsMirrorSourceAddressValid

```
eSM_IsMirrorSourceAddressValid(
tsZCL_ReportAttributeMirror *psZCL_ReportAttributeMirror);
```

Description This function can be used on the ESP to handle mirroring data reported from a Metering Device. If mirroring is enabled, the function should be included in the callback function on the ESP.

When the ESP receives mirroring data from a Metering Device, the event E_ZCL_CBET_ATTRIBUTE_REPORT_MIRROR is generated, causing the callback function to be invoked. The callback function should call this function to deal with the event.

The function first checks that the data comes from a Metering Device which has a mirror on the ESP (the source IEEE address of the data is used for this check) and then updates the event status accordingly:

```
sZCL_CallBackEvent.uMessage.sReportAttributeMirror.eStatus
```

If the source device is valid then this status is set to E_ZCL_ATTR_REPORT_OK and the metering data is automatically stored on the relevant mirror endpoint. Otherwise, a ZCL default response is returned to the Metering Device to indicate that mirroring is not authorised for this device (E_ZCL_CMDS_NOT_AUTHORIZED).

The mirroring of metering data is fully described in Section 42.5.3.

Parameters

- *psZCL_ReportAttributeMirror*: Pointer to sReportAttributeMirror element of the event

Returns

- E_ZCL_SUCCESS

Parent topic: Functions

eSM_ServerUpdateConsumption

```
teZCL_Status eSM_ServerUpdateConsumption(
    uint8 u8SourceEndPointId,
    uint32 u32UtcTime);
```

Description This function can be used on a Simple Metering cluster server (with the 'Get Profile' feature enabled) to add a new entry to the circular buffer used to store historical consumption data. The buffer stores a sequence of entries containing consumption data for consecutive time intervals, identified by their end-times.

Before this function is called, the application must update one or both of the following Simple Metering cluster attributes with the relevant consumption(s) over the last time interval (since the last readings were made):

```
u24CurrentPartialProfileIntervalValueDelivered
u24CurrentPartialProfileIntervalValueReceived
```

An attribute only needs to be updated if the corresponding consumption has been implemented.

The function takes the current time (UTC time) as an input and adds a buffer entry containing the consumption measurements together with the supplied UTC time, which is saved as the end-time of the interval

The entry is stored as a tsSEGetProfile structure, described in Section 42.11.5.

The buffer can contain a limited number of entries, determined at compile-time (see Section 42.12), and operates on a FIFO basis so that a new entry added to a full buffer will over-write the oldest entry.

The function should be called periodically by the application. The period must match the value to which the Simple Metering eProfileIntervalPeriod attribute has been set (see Section 42.2). Standard periods, ranging from 2.5 minutes to one day, are provided as a set of enumerations (see Section 42.10.10).

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which the Simple Metering cluster server operates
- *u32UtcTime*: Current time (as UTC time which can be obtained using **u32ZCL_GetUTCTime()**)

Returns

- E_ZCL_SUCCESS

Parent topic: Functions

eSM_ClientGetProfileCommand

```
teZCL_Status eSM_ClientGetProfileCommand(
    uint8 u8SourceEndpoint,
    uint8 u8DestinationEndpoint,
    tsZCL_Address *psDestinationAddress,
    uint8 u8IntervalChannel,
    uint32 u32EndTime,
    uint8 u8NumberOfPeriods);
```

Description This function can be used on a Simple Metering cluster client (with the 'Get Profile' feature enabled) to send a 'Get Profile' request to the Simple Metering cluster server in order to retrieve historical consumption data.

The server contains a circular buffer which stores a sequence of data entries containing consumption data for consecutive time intervals, identified by their end-times. This function can

request a number of entries from the buffer, containing the consumption data over multiple intervals.

The function parameters include:

- The number of buffer entries (corresponding to consumption intervals) requested
- The most recent end-time for which a buffer entry will be reported - the most recent consumption data will be reported which has an end-time equal to or earlier than this end-time (a specified end-time of zero will result in the most recent consumption data)
- A value indicating whether the units delivered or units received are being requested

This is a non-blocking function and so returns immediately after the request has been sent. The application must then wait for a response, which is accessed using the function **u32SM_GetReceivedProfileData()**.

Parameters

- *u8SourceEndpoint*: Number of local endpoint through which request is sent
- *u8DestinationEndpoint*: Number of endpoint to which request is sent on the destination device
- *psDestinationAddress*: : Pointer to a structure containing the address of the destination device
- *u8IntervalChannel*: Required consumption data - received or delivered:
- E_CLD_SM_CONSUMPTION_RECEIVED
- E_CLD_SM_CONSUMPTION_DELIVERED
- *u32EndTime*: A UTC time representing the most recent interval end-time for which data will be reported (a zero value means report data for the most recent interval)
- *u8NumberOfPeriods*: Number of consumption intervals to be reported

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_CLUSTER_ID_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_EP_RANGE

Parent topic: Functions

u32SM_GetReceivedProfileData

```
uint32 u32SM_GetReceivedProfileData(
    tsSM_GetProfileResponseCommand
    *psSMGetProfileResponseCommand);
```

Description This function is used on a Simple Metering cluster client to obtain the consumption data received in a 'Get Profile' response. The response is obtained from the Simple Metering cluster server (and previously requested by the client through a call to **eSM_ClientGetProfileCommand()**).

When a 'Get Profile' response from the server arrives, the event E_CLD_SM_CLIENT_RECEIVED_COMMAND containing the command E_CLD_SM_GET_PROFILE_RESPONSE is generated on the client. This response causes the callback function on the device to be invoked (for an IPD, this is the callback function registered through **eSE_RegisterIPDEndPoint()**). The callback function should deal with the response.

This function can be called within the callback function to extract consumption data from the event. It is necessary to provide the function with a pointer to the response within the event. The function returns the data corresponding to one consumption interval on each call. Therefore, if the response contains data for multiple intervals, the function must be called multiple times to extract all of this data. The number of intervals contained in the response is also included in the response:

```
sSMCallBackMessage.uMessage.sGetProfileResponseCommand.u8NumberOfPeriodsDelivered
```

When there is no more data to be extracted from the event, the function returns 0xFFFFFFFF.

Parameters

- *psSMGetProfileResponseCommand*: Pointer to sGetProfileResponseCommand element of the event

Returns

- 32-bit value corresponding to consumption data for one interval

0xFFFFFFFF indicates that there is no more data to be read from the event.

Parent topic: Functions

Parent topic:*Simple Metering Cluster*

Return codes The Simple Metering cluster functions use the ZCL return codes, listed in Section 7.2.

Parent topic:*Simple Metering Cluster*

Enumerations

'Attribute ID' Enumerations The following structure contains the enumerations used to identify the attributes of the Simple Metering cluster.

Note: Some of the following enumerations correspond to attributes that are not certifiable in SE 1.1.1 (07-5356-17) or earlier and are for future use (as indicated in the attribute descriptions in Section 42.2).

```
typedef enum PACK
{
    /* Reading information attribute set attribute IDs*/
    E_CLD_SM_ATTR_ID_CURRENT_SUMMATION_DELIVERED    =    0x0000,
    E_CLD_SM_ATTR_ID_CURRENT_SUMMATION_RECEIVED,
    E_CLD_SM_ATTR_ID_CURRENT_MAX_DEMAND_DELIVERED,
```

(continues on next page)

(continued from previous page)

```
    E_CLD_SM_ATTR_ID_CURRENT_MAX_DEMAND_RECEIVED,
    E_CLD_SM_ATTR_ID_DFT_SUMMATION,
    E_CLD_SM_ATTR_ID_DAILY_FREEZE_TIME,
    E_CLD_SM_ATTR_ID_POWER_FACTOR,
    E_CLD_SM_ATTR_ID_READING_SNAPSHOT_TIME,
    E_CLD_SM_ATTR_ID_CURRENT_MAX_DEMAND_DELIVERED_TIME,
    E_CLD_SM_ATTR_ID_CURRENT_MAX_DEMAND_RECEIVED_TIME,
    E_CLD_SM_ATTR_ID_DEFAULT_UPDATE_PERIOD,
    E_CLD_SM_ATTR_ID_FAST_POLL_UPDATE_PERIOD,
    E_CLD_SM_ATTR_ID_CURRENT_BLOCK_PERIOD_CONSUMPTION_DELIVERED,
    E_CLD_SM_ATTR_ID_DAILY_CONSUMPTION_TARGET,
    E_CLD_SM_ATTR_ID_CURRENT_BLOCK,
    E_CLD_SM_ATTR_ID_PROFILE_INTERVAL_PERIOD,
    E_CLD_SM_ATTR_ID_INTERVAL_READ_REPORTING_PERIOD,
    E_CLD_SM_ATTR_ID_PRESET_READING_TIME,
    E_CLD_SM_ATTR_ID_VOLUME_PER_REPORT,
    E_CLD_SM_ATTR_ID_FLOW_RESTRICTION,
    E_CLD_SM_ATTR_ID_SUPPLY_STATUS,
    E_CLD_SM_ATTR_ID_CURRENT_INLET_ENERGY_CARRIER_SUMMATION,
    E_CLD_SM_ATTR_ID_CURRENT_OUTLET_ENERGY_CARRIER_SUMMATION,
    E_CLD_SM_ATTR_ID_INLET_TEMPERATURE,
    E_CLD_SM_ATTR_ID_OUTLET_TEMPERATURE,
    E_CLD_SM_ATTR_ID_CONTROL_TEMPERATURE,
    E_CLD_SM_ATTR_ID_CURRENT_INLET_ENERGY_CARRIER_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_OUTLET_ENERGY_CARRIER_DEMAND,
    /* Time Of Use Information attribute set attribute IDs */
    E_CLD_SM_ATTR_ID_CURRENT_TIER_1_SUMMATION_DELIVERED  =  0x0100,
    E_CLD_SM_ATTR_ID_CURRENT_TIER_1_SUMMATION_RECEIVED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER_2_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER_2_SUMMATION_RECEIVED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER_15_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER_15_SUMMATION_RECEIVED,
    /* Meter status attribute set attribute IDs */
    E_CLD_SM_ATTR_ID_STATUS          =   0x0200,
    E_CLD_SM_ATTR_ID_REMAINING_BATTERY_LIFE,
    E_CLD_SM_ATTR_ID_HOURS_IN_OPERATION,
    E_CLD_SM_ATTR_ID_HOURS_IN_FAULT,
    /* Formatting attribute set attribute IDs */
    E_CLD_SM_ATTR_ID_UNIT_OF_MEASURE          =   0x0300,
    E_CLD_SM_ATTR_ID_MULTIPLIER,
    E_CLD_SM_ATTR_ID_DIVISOR,
    E_CLD_SM_ATTR_ID_SUMMATION_FORMATING,
    E_CLD_SM_ATTR_ID_DEMAND_FORMATING,
    E_CLD_SM_ATTR_ID_HISTORICAL_CONSUMPTION_FORMATTING,
    E_CLD_SM_ATTR_ID_METERING_DEVICE_TYPE,
    E_CLD_SM_ATTR_ID_SITE_ID,
    E_CLD_SM_ATTR_ID_METER_SERIAL_NUMBER,
    E_CLD_SM_ATTR_ID_ENERGY_CARRIER_UNIT_OF_MEASURE,
    E_CLD_SM_ATTR_ID_ENERGY_CARRIER_SUMMATION_FORMATTING,
    E_CLD_SM_ATTR_ID_ENERGY_CARRIER_DEMAND_FORMATTING,
    E_CLD_SM_ATTR_ID_TEMPERATURE_UNIT_OF_MEASURE,
    E_CLD_SM_ATTR_ID_TEMPERATURE_FORMATTING,
    /* ESP Historical Consumption set attribute IDs */
    E_CLD_SM_ATTR_ID_INSTANTANEOUS_DEMAND        =   0x0400,
    E_CLD_SM_ATTR_ID_CURRENT_DAY_CONSUMPTION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_DAY_CONSUMPTION_RECEIVED,
    E_CLD_SM_ATTR_ID_PREVIOUS_DAY_CONSUMPTION_DELIVERED,
    E_CLD_SM_ATTR_ID_PREVIOUS_DAY_CONSUMPTION_RECEIVED,
    E_CLD_SM_ATTR_ID_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_
↪DELIVERED,
```

(continues on next page)

(continued from previous page)

```
    E_CLD_SM_ATTR_ID_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_RECEIVED,
    E_CLD_SM_ATTR_ID_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_RECEIVED,
    E_CLD_SM_ATTR_ID_CURRENT_DAY_MAXIMUM_PRESSURE,
    E_CLD_SM_ATTR_ID_CURRENT_DAY_MINIMUM_PRESSURE,
    E_CLD_SM_ATTR_ID_PREVIOUS_DAY_MAXIMUM_PRESSURE,
    E_CLD_SM_ATTR_ID_PREVIOUS_DAY_MINIMUM_PRESSURE,
    E_CLD_SM_ATTR_ID_CURRENT_DAY_MAXIMUM_DEMAND,
    E_CLD_SM_ATTR_ID_PREVIOUS_DAY_MAXIMUM_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_MONTH_MAXIMUM_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_YEAR_MAXIMUM_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_DAY_MAXIMUM_ENERGY_CARRIER_DEMAND,
    E_CLD_SM_ATTR_ID_PREVIOUS_DAY_MAXIMUM_ENERGY_CARRIER_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_MONTH_MAXIMUM_ENERGY_CARRIER_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_MONTH_MINIMUM_ENERGY_CARRIER_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_YEAR_MAXIMUM_ENERGY_CARRIER_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_YEAR_MINIMUM_ENERGY_CARRIER_DEMAND,
    /* Load Profile attribute set attribute IDs */
    E_CLD_SM_ATTR_ID_MAX_NUMBER_OF_PERIODS_DELIVERED     =    0x0500,
    /* Supply Limit attribute set attribute IDs */
    E_CLD_SM_ATTR_ID_CURRENT_DEMAND_DELIVERED     =    0x0600,
    E_CLD_SM_ATTR_ID_DEMAND_LIMIT,
    E_CLD_SM_ATTR_ID_DEMAND_INTEGRATION_PERIOD,
    E_CLD_SM_ATTR_ID_NUMBER_OF_DEMAND_SUBINTERVALS,
    /* Block Information Attribute set attribute IDs */
    /* Block Information Attribute set: No Tier Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_NOTIER_BLOCK1_SUMMATION_DELIVERED  =   0x0700,
    E_CLD_SM_ATTR_ID_CURRENT_NOTIER_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_NOTIER_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier1 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER1_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER1_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER1_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier2 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER2_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER2_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER2_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier5 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER3_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER3_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER3_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier4 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER4_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER4_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER4_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier5 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER5_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER5_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER5_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier6 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER6_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER6_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER6_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier8 Block Set */
```

(continues on next page)

(continued from previous page)

```
    E_CLD_SM_ATTR_ID_CURRENT_TIER7_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER7_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER7_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier8 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER8_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER8_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER8_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier9 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER9_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER9_BLOCK2_SUMMATION_DELIVERED,
    :
    E_CLD_SM_ATTR_ID_CURRENT_TIER9_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier10 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER10_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER10_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER10_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier11 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER11_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER11_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER11_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier12 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER12_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER12_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER12_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier13 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER13_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER13_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER13_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier14 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER14_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER14_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER14_BLOCK16_SUMMATION_DELIVERED,
    /* Block Information Attribute set: Tier15 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER15_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER15_BLOCK2_SUMMATION_DELIVERED,
:
    E_CLD_SM_ATTR_ID_CURRENT_TIER15_BLOCK16_SUMMATION_DELIVERED,
    /* Alarm Attribute set attribute IDs */
    E_CLD_SM_ATTR_ID_GENERIC_ALARM_MASK = 0x0800,
    E_CLD_SM_ATTR_ID_ELECTRICITY_ALARM_MASK,
    E_CLD_SM_ATTR_ID_PRESSURE_ALARM_MASK,
    E_CLD_SM_ATTR_ID_WATER_SPECIFIC_ALARM_MASK,
    E_CLD_SM_ATTR_ID_HEAT_AND_COOLING_SPECIFIC_ALARM_MASK,
    E_CLD_SM_ATTR_ID_GAS_ALARM_MASK,
} teCLD_SM_SimpleMeteringAttributeID;
```

Parent topic: Enumerations

'Meter Status' Enumerations Enumerations for the u8MeterStatus element in the Simple Metering cluster structure tsSE_SimpleMetering are provided as #defines.

The following enumerated masks can be used to set the meter status:

Enumeration	Description
E_CLD_SM_METER_STATUS_CHECK_METER_MASK	Non-fatal problem detected on meter
E_CLD_SM_METER_STATUS_LOW_BATTERY_MASK	Battery level is low
E_CLD_SM_METER_STATUS_TAMPER_DETECT_MASK	Detected tampering with device
E_CLD_SM_METER_STATUS_POWER_FAILURE_MASK	Indicates power failure on device
E_CLD_SM_METER_STATUS_POWER_QUALITY_MASK	Power anomaly detected
E_CLD_SM_METER_STATUS_LEAK_DETECT_MASK	Detected leak (e.g. of gas or water)
E_CLD_SM_METER_STATUS_SERVICE_DISCONNECT_OPEN_	Service to premises disconnected

Parent topic: Enumerations

'Unit of Measure' Enumerations The following enumerations are used to set the teSE_UnitOfMeasure element in the Simple Metering cluster structure tsSE_SimpleMetering. Separate sets of enumerations are provided for binary and BCD (Binary Coded Decimal) representations.

```
typedef enum PACK
{
    /* Binary values */
    E_CLD_SM_UOM_KILO_WATTS                = 0x00,
    E_CLD_SM_UOM_CUBIC_METER,
    E_CLD_SM_UOM_CUBIC_FEET,
    E_CLD_SM_UOM_100_CUBIC_FEET,           /* ccf & ccf/h */
    E_CLD_SM_UOM_US_GALLON,                /* USG & USG/h */
    E_CLD_SM_UOM_IMPERIAL_GALLON,          /* IMPG & IMPG/h */
    E_CLD_SM_UOM_BTU,                      /* BTU & BTU/h */
    E_CLD_SM_UOM_LITERS,                   /* Liters & Liters/h */
    E_CLD_SM_UOM_KPA_GAUGE,
    E_CLD_SM_UOM_KPA_ABSOLUTE,
    /* BCD values */
    E_CLD_SM_UOM_KILO_WATTS_BCD            = 0x80,
    E_CLD_SM_UOM_CUBIC_METER_BCD,
    E_CLD_SM_UOM_CUBIC_FEET_BCD,
    E_CLD_SM_UOM_100_CUBIC_FEET_BCD,       /* ccf & ccf/h */
    E_CLD_SM_UOM_US_GALLON_BCD,            /* USG & USG/h */
    E_CLD_SM_UOM_IMPERIAL_GALLON_BCD,      /* IMPG & IMPG/h */
    E_CLD_SM_UOM_BTU_BCD,                  /* BTU & BTU/h */
    E_CLD_SM_UOM_LITERS_BCD,               /* Liters & Liters/h */
    E_CLD_SM_UOM_KPA_GAUGE_BCD,
    E_CLD_SM_UOM_KPA_ABSOLUTE_BCD
} teCLD_SM_UnitOfMeasure;
```

The above enumerations are detailed in the table below.

Enumeration	Description
	Instantaneous
Binary Values	
E_CLD_SM_UOM_KILO_WATTS	kW (kiloWatts)
E_CLD_SM_UOM_CUBIC_METER	m3/h (cubic metres per hour)
E_CLD_SM_UOM_CUBIC_FEET	ft3/h (cubic feet per hour)
E_CLD_SM_UOM_100_CUBIC_FEET	ccf/h (100 cubic feet per hour)
E_CLD_SM_UOM_US_GALLON	US gl/h (US Gallons per hour)
E_CLD_SM_UOM_IMPERIAL_GALLON	Imperial gl/h (Imperial Gallons per hour)
E_CLD_SM_UOM_BTU	BTU/h (British Thermal Units per hour)
E_CLD_SM_UOM_LITERS	l/h (litres per hour)
E_CLD_SM_UOM_KPA_GAUGE	kPA (kiloPascal) gauge
E_CLD_SM_UOM_KPA_ABSOLUTE	kPA (kiloPascal) absolute
BCD Values	
E_CLD_SM_UOM_KILO_WATTS_BCD	kW (kiloWatts)
E_CLD_SM_UOM_CUBIC_METER_BCD	m3/h (cubic metres per hour)
E_CLD_SM_UOM_CUBIC_FEET_BCD	ft3/h (cubic feet per hour)
E_CLD_SM_UOM_100_CUBIC_FEET_BCD	ccf/h (100 cubic feet per hour)
E_CLD_SM_UOM_US_GALLON_BCD	US gl/h (US Gallons per hour)
E_CLD_SM_UOM_IMPERIAL_GALLON_BCD	Imperial gl/h (Imperial Gallons per hour)
E_CLD_SM_UOM_BTU_BCD	BTU/h (British Thermal Units per hour)
E_CLD_SM_UOM_LITERS_BCD	l/h (litres per hour)
E_CLD_SM_UOM_KPA_GAUGE_BCD	kPA (kiloPascal) gauge
E_CLD_SM_UOM_KPA_ABSOLUTE_BCD	kPA (kiloPascal) absolute

Parent topic: Enumerations

'Summation Formatting' Enumerations Enumerations for use with the u8SummationFormatting element in the Simple Metering cluster structure u8SummationFormatting are provided as #defines. The enumerations allow the following formatting information to be extracted from the u8SummationFormatting bitmap:

Enumeration	Description
E_CLD_SM_FORMATTING_DIGITS_TO_RI	Position of least significant bit of bit-field indicating number of digits to right of decimal point
E_CLD_SM_FORMATTING_DIGITS_TO_RI	Number of bits in bit-field indicating number of digits to right of decimal point
E_CLD_SM_FORMATTING_DIGITS_TO_RI	Bit-mask used to extract number of digits to right of decimal point
E_CLD_SM_FORMATTING_DIGITS_TO_LE	Position of least significant bit of bit-field indicating number of digits to left of decimal point
E_CLD_SM_FORMATTING_DIGITS_TO_LE	Number of bits in bit-field indicating number of digits to left of decimal point
E_CLD_SM_FORMATTING_DIGITS_TO_LE	Bit-mask used to extract number of digits to left of decimal point
E_CLD_SM_FORMATTING_SUPPRESS_LE	Bit-mask used to extract bit indicating whether lead-ing zeros will be suppressed

The following are examples of the use of the above enumerations.

Extracting the number of digits to the right of the decimal point:

```
u8BitsToRight = (u8SummationFormatting & E_CLD_SM_FORMATTING_DIGITS_TO_RIGHT_OF_
↪DP_MASK)
>> E_CLD_SM_FORMATTING_DIGITS_TO_RIGHT_OF_DP_LS_BIT
```

Extracting the number of digits to the left of the decimal point:

```
u8BitsToLeft = (u8SummationFormatting & E_CLD_SM_FORMATTING_DIGITS_TO_LEFT_OF_DP_
↪MASK)
>> E_CLD_SM_FORMATTING_DIGITS_TO_LEFT_OF_DP_LS_BIT
```

Determining whether leading zeros will be suppressed:

```
bSuppressZeros = !((u8SummationFormatting & E_CLD_SM_FORMATTING_SUPPRESS_LEADING_
↪ZEROS_BIT) == 0)
```

Parent topic: Enumerations

'Supply Direction' Enumerations The following enumerations are used to indicate the direction of supply.

```
typedef enum PACK
{
    E_CLD_SM_CONSUMPTION_DELIVERED,
    E_CLD_SM_CONSUMPTION_RECEIVED
}teSM_IntervalChannel;
```

The above enumerations are detailed in the table below.

Enumeration	Description
E_CLD_SM_CONSUMPT	Specifies that the supply is from the customer to the utility company (in cases where the customer generates their own supply)
E_CLD_SM_CONSUMPT	Specifies that the supply is from the utility company to the customer

Parent topic: Enumerations

'Metering Device Type' Enumerations The following enumerations are used to set the eMeteringDeviceType element in the Simple Metering cluster structure tsSE_SimpleMetering.

```
typedef enum PACK
{
    E_CLD_SM_MDT_ELECTRIC                = 0x00,
    E_CLD_SM_MDT_GAS,
    E_CLD_SM_MDT_WATER,
    E_CLD_SM_MDT_THERMAL,                /* Deprecated */
    E_CLD_SM_MDT_PRESSURE,
    E_CLD_SM_MDT_HEAT,
    E_CLD_SM_MDT_COOLING,
    E_CLD_SM_MDT_GAS_MIRRORED            = 0x80,
    E_CLD_SM_MDT_WATER_MIRRORED,
    E_CLD_SM_MDT_THERMAL_MIRRORED,
    E_CLD_SM_MDT_PRESSURE_MIRRORED,
```

(continues on next page)

(continued from previous page)

```
    E_CLD_SM_MDT_HEAT_MIRRORED,
    E_CLD_SM_MDT_COOLING_MIRRORED,
} teCLD_SM_MeteringDeviceType;
```

The above enumerations are detailed in the table below.

Enumeration	Description
E_CLD_SM_MDT_ELECTRIC	Electric Meter
E_CLD_SM_MDT_GAS	Gas Meter
E_CLD_SM_MDT_WATER	Water Meter
E_CLD_SM_MDT_THERMAL	Thermal Meter (deprecated)
E_CLD_SM_MDT_PRESSURE	Pressure Meter
E_CLD_SM_MDT_HEAT	Heat Meter
E_CLD_SM_MDT_COOLING	Cooling Meter
E_CLD_SM_MDT_GAS_MIRRORED	Mirrored Gas Meter
E_CLD_SM_MDT_WATER_MIRRORED	Mirrored Water Meter
E_CLD_SM_MDT_THERMAL_MIRRORED	Mirrored Thermal Meter (deprecated)
E_CLD_SM_MDT_PRESSURE_MIRRORED	Mirrored Pressure Meter
E_CLD_SM_MDT_HEAT_MIRRORED	Mirrored Heat Meter
E_CLD_SM_MDT_COOLING_MIRRORED	Mirrored Cooling Meter

Parent topic: Enumerations

'Simple Metering Event' Enumerations The event types generated by the Simple Metering cluster are enumerated in the teSM_CallBackEventType structure below:

```
typedef enum PACK
{
        E_CLD_SM_CLIENT_RECEIVED_COMMAND,
        E_CLD_SM_SERVER_RECEIVED_COMMAND,
        E_CLD_SM_FAST_POLLING_TIMER_EXPIRED
}teSM_CallBackEventType;
```

The above event types are described in the table below.

Event Type Enumeration	Description
E_CLD_SM_CLIENT_RECEIVE	Generated on a cluster client when a command is received from the server
E_CLD_SM_SERVER_RECEIVE	Generated on the cluster server when a command is received from a client
E_CLD_SM_FAST_POLLING_T	Generated on the cluster server when the end-time of a fast polling episode is reached (*for future use*)

Parent topic: Enumerations

'Server Command' Enumerations The comands issued by a Simple Metering cluster server and received by a client are enumerated in the teSM_ClusterServerCommands structure below:

```
typedef enum PACK
{
    E_CLD_SM_GET_PROFILE_RESPONSE,
```

(continues on next page)

(continued from previous page)

```
    E_CLD_SM_REQUEST_MIRROR,
    E_CLD_SM_REMOVE_MIRROR,
    E_CLD_SM_REQUEST_FAST_POLL_MODE_RESPONSE,
    E_CLD_SM_CLIENT_ERROR
}teSM_ClusterServerCommands;
```

Command Enumeration	Description
E_CLD_SM_GET_PROFILI	Response to 'Get Profile' request - content of response is contained in the structure tsS-M_GetProfileResponseCommand in the event (see Section 42.11.9)
E_CLD_SM_REQUEST_MI	An 'Add Mirror' request
E_CLD_SM_REMOVE_MII	A 'Remove Mirror' request
E_CLD_SM_REQUEST_FA RESPONSE	Response to 'Fast Polling' request (*for future use*)
E_CLD_SM_CLIENT_ERR	Error condition - content of error is contained in the structure tsSM_Error in the event (see Section 42.11.10)

Parent topic: Enumerations

'Client Command' Enumerations The comands issued by a Simple Metering cluster client and received by the server are enumerated in the teSM_ClusterClientCommands structure below:

```
typedef enum PACK
{
    E_CLD_SM_GET_PROFILE,
    E_CLD_SM_REQUEST_MIRROR_RESPONSE,
    E_CLD_SM_MIRROR_REMOVED,
    E_CLD_SM_REQUEST_FAST_POLL_MODE,
    E_CLD_SM_SERVER_ERROR
}teSM_ClusterClientCommands;
```

Command Enumeration	Description
E_CLD_SM_GET_PR	A 'Get Profile' request - content of request is contained in the structure tsSM_GetProfile-RequestCommand in the event (see Section42.11.8)
E_CLD_SM_REQUES	Response to 'Add Mirror' request - content of response is contained in the structure tsS-M_RequestMirrorResponseCommand in the event (see Section 42.11.6)
E_CLD_SM_MIRROR	Response to 'Remove Mirror' request - content of response is contained in the structure tsS-M_MirrorRemovedResponseCommand in the event (see Section 42.11.7)
E_CLD_SM_REQUES	A 'Fast Polling' request (*for future use*)
E_CLD_SM_SERVER_	Error condition - content of error is contained in the structure tsSM_Error in the event (see Section 42.11.10)

Parent topic: Enumerations

'Consumption Interval' Enumerations The following enumerations define the possible time-intervals for the consumption data captured in the 'Get Profile' feature.

```
typedef enum PACK
{
    E_CLD_SM_TIME_FRAME_DAILY,
    E_CLD_SM_TIME_FRAME_60MINS,
    E_CLD_SM_TIME_FRAME_30MINS,
    E_CLD_SM_TIME_FRAME_15MINS,
    E_CLD_SM_TIME_FRAME_10MINS,
    E_CLD_SM_TIME_FRAME_7_5MINS,
    E_CLD_SM_TIME_FRAME_5MINS,
    E_CLD_SM_TIME_FRAME_2_5MINS
}teSM_TimeFrame;
```

Time Frame Enumeration	Time Interval
E_CLD_SM_TIME_FRAME_DAILY	One day
E_CLD_SM_TIME_FRAME_60MINS	60 minutes
E_CLD_SM_TIME_FRAME_30MINS	30 minutes
E_CLD_SM_TIME_FRAME_15MINS	15 minutes
E_CLD_SM_TIME_FRAME_10MINS	10 minutes
E_CLD_SM_TIME_FRAME_7_5MINS	7.5 minutes
E_CLD_SM_TIME_FRAME_5MINS	5 minutes
E_CLD_SM_TIME_FRAME_2_5MINS	2.5 minutes

Parent topic: Enumerations

'Simple Metering Status' Enumerations The following enumerations are used to report status in the Simple Metering cluster.

```
typedef enum PACK
{
    E_CLD_SM_STATUS_SUCCESS,
    E_CLD_SM_STATUS_UNDEFINED_INTERVAL_CHANNEL,
    E_CLD_SM_STATUS_INTERVAL_NOT_SUPPORTED,
    E_CLD_SM_STATUS_INVALID_END_TIME,
    E_CLD_SM_STATUS_MORE_PERIODS_REQUESTED_THAN_SUPPORTED,
    E_CLD_SM_STATUS_NO_INTERVALS_AVAILABLE_FOR_REQUESTED_TIME,
    E_CLD_SM_STATUS_EP_NOT_AVAILABLE
}teSM_Status;
```

Status Enumeration	Description
E_CLD_SM_STATUS_SUCCESS	Success
E_CLD_SM_STATUS_UNDEFINED_INTI	Undefined eIntervalChannel value speci-fied in 'Get Profile' request (see Section42.11.8)
E_CLD_SM_STATUS_INTERVAL_NOT_S	Unsupported consumption data specifed through eIntervalChannel in 'Get Profile' request (see Section 42.11.8)
E_CLD_SM_STATUS_INVALID_END_TI	Invalid end-time specified in 'Get Profile' request (Section 42.11.8)
E_CLD_SM_STATUS_MORE_PERIODS_	More periods specified in 'Get Profile' request than can be returned
E_CLD_SM_STATUS_NO_INTERVALS_A	No intervals available for the end-time specified in 'Get Profile' request
E_CLD_SM_STATUS_EP_NOT_AVAILAE	Specified endpoint not available

Parent topic: Enumerations

Parent topic:*Simple Metering Cluster*

Structures

tsSM_CallBackMessage For a Simple Metering event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsSM_CallBackMessage structure which contains the Simple Metering parameters:

```
typedef struct
{
    teSM_CallBackEventType eEventType;
     uint8 u8CommandId;
  union
    {
    tsSM_GetProfileResponseCommand     sGetProfileResponseCommand;
    tsSM_RequestFastPollResponseCommand  sRequestFastPollResponseCommand;
    tsSM_GetProfileRequestCommand     sGetProfileCommand;
    tsSM_RequestMirrorResponseCommand    sRequestMirrorResponseCommand;
    tsSM_MirrorRemovedResponseCommand    sMirrorRemovedResponseCommand;
    tsSM_RequestFastPollCommand      sRequestFastPollCommand;
    tsSM_Error            sError;
 }uMessage;
}tsSM_CallBackMessage;
```

where:

- eEventType is the Simple Metering event type from those listed in Section42.10.7
- u8CommandId is the identifier of the type of Simple Metering command received. This field is only valid for the following Simple Metering event types:
  - E_CLD_SM_CLIENT_RECEIVED_COMMAND - enumerated commands are provided, as described in Section 42.10.8
  - E_CLD_SM_SERVER_RECEIVED_COMMAND - enumerated commands are provided, as described in Section 42.10.9
- uMessage is a union containing the command payload in one of the following forms (depending on the command specified in the field u8CommandId):
  - s``GetProfileResponseCommand is a structure containing the payload of a 'Get Profile' response - see Section 42.11.9
  - sRequestFastPollResponseCommand is a structure containing the payload of a 'Fast Polling' response (*for future use*)
  - sGetProfileCommand is a structure containing the payload of a 'Get Profile' request - see Section 42.11.8
  - sRequestMirrorResponseCommand is a structure containing the payload of an 'Add Mirror' response - see Section 42.11.6
  - sMirrorRemovedResponseCommand is a structure containing the payload of an 'Remove Mirror' response - see Section 42.11.7
  - sRequestFastPollCommand is a structure containing the payload of an 'Fast Polling' request (*for future use*)
  - sError is a structure containing the details of an error condition - see Section 42.11.10

Parent topic: Structures

tsSE_Mirror Details of the mirror endpoints on the ESP are kept in an array of structures of the type tsSE_Mirror (one structure per endpoint) within the tsSE_EspMeterDevice structure. The tsSE_Mirror structure is shown and described below.

Note: This structure is only for use by the ZCL and should not be modified by the application.

```
typedef struct
{
            /*Mirrored EndPoint*/
    tsZCL_EndPointDefinition sEndPoint;
    /*Mirror Requester address*/
    uint64        u64SourceAddress;
    /*Mirror cluster instances*/
    tsSE_MirrorClusterInstances          sSEMirrorClusterInstances;
    /*Event Address, Custom callback event, Custom callback message*/
    tsSM_CustomStruct sSMMirrorCustomDataStruct;
}tsSE_Mirror;
```

where:

- sEndPoint is a tsZCL_EndPointDefinition structure which contains details of the endpoint corresponding to the mirror (for details of this structure, refer to Section 6.1.1)
- u64SourceAddress is the 64-bit IEEE address of the Metering Device to which the mirror endpoint is assigned - a zero value indicates that the mirror endpoint is not currently assigned to a device
- sSEMirrorClusterInstances is a tsSE_MirrorClusterInstances structure (see Section 42.11.3) which contains information on the Basic and Simple Metering cluster instances that are associated with the mirror endpoint
- sSMMirrorCustomDataStruct is a tsSM_CustomStruct structure (see Section 42.11.4) which contains data relating to a received command/message for the mirror endpoint

Parent topic: Structures

tsSE_MirrorClusterInstances This structure contains information on the Basic and Simple Metering cluster instances that are associated with a mirror endpoint.

Note: This structure is only for use by the ZCL and should not be modified by the application.

```
typedef struct
{
   /*Basic Cluster Instance*/
          tsZCL_ClusterInstance sBasicCluster;
          /* SM Cluster Instance */
          tsZCL_ClusterInstance sSM_Cluster;
}tsSE_MirrorClusterInstances;
```

where:

- sBasicCluster is a tsZCL_ClusterInstance structure which contains information on the Basic cluster instance associated with a mirror endpoint (for details of this structure, refer to Section 6.1.16)
- sSM_Cluster is a tsZCL_ClusterInstance structure which contains information on the Simple Metering cluster instance associated with a mirror endpoint (for details of this structure, refer to Section 6.1.16)

Parent topic: Structures

tsSM_CustomStruct This structure contains data relating to a command/message for a mirror endpoint.

Note: This structure is only for use by the ZCL software and should not be modified by the application.

```
typedef struct
{
    tsZCL_ReceiveEventAddress  sReceiveEventAddress;
    tsZCL_CallBackEvent        sSMCustomCallBackEvent;
    tsSM_CallBackMessage       sSMCallBackMessage;
} tsSM_CustomStruct;
```

where:

- sReceiveEventAddress is a tsZCL_ReceiveEventAddress structure which contains addressing information relating to a received mirroring command/message
- sSMCustomCallBackEvent is a tsZCL_CallBackEvent structure (see Section 3.1) which contains the event that has been generated as a result of the received command/message
- sSMCallBackMessage is a tsSM_CallBackMessage structure (see Section42.11.1) which contains details of the event and the command/message that caused the event

Parent topic: Structures

tsSEGetProfile This structure is used to store historical consumption data when the 'Get Profile' feature is enabled. The data within the structure corresponds to a single consumption interval.

```
typedef struct
{
        uint32 u32UtcTime;
        zuint24 u24ConsumptionReceived;
        zuint24 u24ConsumptionDelivered;
}tsSEGetProfile;
```

where:

- u32UtcTime is the end-time of the consumption interval (as a UTC time)
- u24ConsumptionReceived is the number of units received from the customer during the interval (for customers who generate and sell their own units)
- u24ConsumptionDelivered is the number of units delivered to the customer during the interval

Parent topic: Structures

tsSM_RequestMirrorResponseCommand This structure contains the details of an 'Add Mirror' response (from a cluster client). It is included in the structure tsSM_CallBackMessage when an E_CLD_SM_SERVER_RECEIVED_COMMAND event containing the command E_CLD_SM_REQUEST_MIRROR_RESPONSE is generated on the cluster server.

```
typedef struct
{
    uint16 u16Endpoint;
}tsSM_RequestMirrorResponseCommand;
```

where u16Endpoint is the number of the endpoint on which the mirror was successfully added or takes the value 0xFFFF if the request failed because no free endpoint was available for the mirror.

Parent topic: Structures

tsSM_MirrorRemovedResponseCommand This structure contains the details of a 'Remove Mirror' response (from a cluster client). It is included in the structure tsSM_CallBackMessage when an E_CLD_SM_SERVER_RECEIVED_COMMAND event containing the command E_CLD_SM_MIRROR_REMOVED is generated on the cluster server.

```
typedef struct
{
    uint16 u16Endpoint;
}tsSM_MirrorRemovedResponseCommand;
```

where u16Endpoint is the number of the endpoint from which the mirror was successfully removed, or takes the value 0xFFFF if the remove request failed.

Parent topic: Structures

tsSM_GetProfileRequestCommand This stucture contains the details of a 'Get Profile' request (from a cluster client). It is included in the structure tsSM_CallBackMessage when an E_CLD_SM_SERVER_RECEIVED_COMMAND event containing the command E_CLD_SM_GET_PROFILE is generated on the cluster server.

```
typedef struct
{
    teSM_IntervalChannel eIntervalChannel;
    uint8 u8NumberOfPeriods;
    uint8 u8SourceEndPoint;
    uint8 u8DestinationEndPoint;
    uint32 u32EndTime;
    tsZCL_Address sSourceAddress;
}tsSM_GetProfileRequestCommand;
```

where:

- eIntervalChannel is a value indicating the required consumption data:
  - E_CLD_SM_CONSUMPTION_RECEIVED - units from customer
  - E_CLD_SM_CONSUMPTION_DELIVERED - units to customer
- u8NumberOfPeriods is the number of consumption intervals for which data is being requested
- u8SourceEndPoint is the number of the source endpoint of the request on the client
- u8DestinationEndPoint is the number of the destination endpoint of the request on the server
- u32EndTime is the end-time for which consumption data is being requested - the most recent consumption data will be reported which has an end-time equal to or earlier than this end-time (a zero value will result in the most recent consumption data)
- sSourceAddress is a structure containing the source address of the request - that is, the address of the requesting client (the structure is described in Section 6.1.4)

Parent topic: Structures

tsSM_GetProfileResponseCommand This stucture contains the details of a 'Get Profile' response (from the cluster server). It is included in the structure tsSM_CallBackMessage when an E_CLD_SM_CLIENT_RECEIVED_COMMAND event containing the command E_CLD_SM_GET_PROFILE_RESPONSE is generated on the cluster server.

```
typedef struct
{
    uint32          u32Endtime;
    teSM_Status     eStatus;
    teSM_TimeFrame u8ProfileIntervalPeriod;
    uint8           u8NumberOfPeriodsDelivered;
    zuint24         *pau24Intervals;
}tsSM_GetProfileResponseCommand;
```

where:

- u32Endtime is the end-time of the consumption data that is being reported, as a UTC time
- eStatus is the status of the response, represented by one of the enumerated values listed in Section 42.10.11
- u8ProfileIntervalPeriod is the time-interval (consumption interval) over which each set of consumption data is collected - one of the standard enumerated values listed in Section 42.10.10
- u8NumberOfPeriodsDelivered is the number of consumption intervals being reported
- pau24Intervals is a pointer to the consumption data being reported

Parent topic: Structures

tsSM_Error This stucture contains the details of an error response (from cluster server or client). It is included in the structure tsSM_CallBackMessage when an E_CLD_SM_SERVER_RECEIVED_COMMAND event is generated containing the command E_CLD_SM_SERVER_ERROR on a client or E_CLD_SM_CLIENT_ERROR on the server.

```
typedef struct
{
    uint8 u8Endpoint;
    uint8 u8Status;
}tsSM_Error;
```

where

- u8Endpoint is the number of the endpoint from which the error is reported
- u8Status is a value representing the nature of the error

Parent topic: Structures

Parent topic:*Simple Metering Cluster*

Compile-time options This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the Simple Metering cluster.

The Simple Metering cluster is enabled by defining CLD_SIMPLE_METERING.

Optional Attributes The optional attributes for the Simple Metering cluster are enabled/disabled by defining:

- For optional attributes from 'Reading Information' attribute set:

  - CLD_SM_ATTR_CURRENT_SUMMATION_RECEIVED
  - CLD_SM_ATTR_CURRENT_MAX_DEMAND_DELIVERED
  - CLD_SM_ATTR_CURRENT_MAX_DEMAND_RECEIVED
  - CLD_SM_ATTR_DFT_SUMMATION
  - CLD_SM_ATTR_DAILY_FREEZE_TIME
  - CLD_SM_ATTR_POWER_FACTOR
  - CLD_SM_ATTR_READING_SNAPSHOT_TIME
  - CLD_SM_ATTR_CURRENT_MAX_DEMAND_DELIVERED_TIME
  - CLD_SM_ATTR_CURRENT_MAX_DEMAND_RECEIVED_TIME
- For optional attributes from 'Time-Of-Use (TOU) Information' attribute set:
  - CLD_SM_ATTR_CURRENT_TIER_1_SUMMATION_DELIVERED
  - CLD_SM_ATTR_CURRENT_TIER_1_SUMMATION_RECEIVED
  - CLD_SM_ATTR_CURRENT_TIER_2_SUMMATION_DELIVERED
  - CLD_SM_ATTR_CURRENT_TIER_2_SUMMATION_RECEIVED
  - CLD_SM_ATTR_CURRENT_TIER_3_SUMMATION_DELIVERED
  - CLD_SM_ATTR_CURRENT_TIER_3_SUMMATION_RECEIVED
  - CLD_SM_ATTR_CURRENT_TIER_4_SUMMATION_DELIVERED
  - CLD_SM_ATTR_CURRENT_TIER_4_SUMMATION_RECEIVED
  - CLD_SM_ATTR_CURRENT_TIER_5_SUMMATION_DELIVERED
  - CLD_SM_ATTR_CURRENT_TIER_5_SUMMATION_RECEIVED
  - CLD_SM_ATTR_CURRENT_TIER_6_SUMMATION_DELIVERED
  - CLD_SM_ATTR_CURRENT_TIER_6_SUMMATION_RECEIVED
- For optional attributes from 'Block Information' attribute set:
  - CLD_SM_ATTR_NO_TIER_BLOCK_CURRENT_SUMMATION_DELIVERED_MAX_COUNT (maximum value of 16)
  - CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED

    (maximum value of 15)
  - CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED

    (maximum value of 16)
- For optional attributes from 'Formatting' attribute set:
  - CLD_SM_ATTR_MULTIPLIER
  - CLD_SM_ATTR_DIVISOR
  - CLD_SM_ATTR_DEMAND_FORMATING
  - CLD_SM_ATTR_HISTORICAL_CONSUMPTION_FORMATTING
- For optional attributes from 'ESP Historical Consumption' attribute set:
  - CLD_SM_ATTR_INSTANTANEOUS_DEMAND
  - CLD_SM_ATTR_CURRENT_DAY_CONSUMPTION_DELIVERED
  - CLD_SM_ATTR_CURRENT_DAY_CONSUMPTION_RECEIVED
  - CLD_SM_ATTR_PREVIOUS_DAY_CONSUMPTION_DELIVERED

  - CLD_SM_ATTR_PREVIOUS_DAY_CONSUMPTION_RECEIVED
  - CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_DELIVERED
  - CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_RECEIVED
  - CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_DELIVERED
  - CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_RECEIVED
- For optional attribute from 'Load Profile' attribute set:
  - CLD_SM_ATTR_MAX_NUMBER_OF_PERIODS_DELIVERED
- For optional attributes from 'Supply Limit' attribute set:
  - CLD_SM_ATTR_CURRENT_DEMAND_DELIVERED
  - CLD_SM_ATTR_DEMAND_LIMIT
  - CLD_SM_ATTR_DEMAND_INTEGRATION_PERIOD
  - CLD_SM_ATTR_NUMBER_OF_DEMAND_SUBINTERVALS

Mirroring If the mirroring of metering data is to be enabled (see Section 42.5), the following options must be defined in the **zcl_options.h** file.

On the Simple Metering server on the Metering Device (which will request and report to a mirror on a mirroring server, such as the ESP), there is no need to define anything.

On the Simple Metering client on the mirroring server, such as the ESP, the mirroring option must be enabled by including:

```
#define CLD_SM_SUPPORT_MIRROR
```

In addition, the following defines must be added on the mirroring server (e.g. ESP):

```
#define CLD_BAS_ATTR_PHYSICAL_ENVIRONMENT
```

(flags support for mirroring via a non-zero value of the u8PhysicalEnvironment attribute of the Basic cluster)

```
#define CLD_SM_NUMBER_OF_MIRRORS          <n>
```

(sets the maximum number of mirrors supported on the mirroring server to the value n)

```
#define ZCL_ATTRIBUTE_REPORTING_CLIENT_SUPPORTED
```

(enables support for attribute reporting clients)

The Simple Metering cluster attributes that are supported by mirroring must be defined on the mirroring server (the same set of attributes are mirrored on all endpoints):

- CLD_SM_MIRROR_ATTR_CURRENT_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_MAX_DEMAND_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_MAX_DEMAND_RECEIVED
- CLD_SM_MIRROR_ATTR_DFT_SUMMATION
- CLD_SM_MIRROR_ATTR_DAILY_FREEZE_TIME
- CLD_SM_MIRROR_ATTR_POWER_FACTOR

- CLD_SM_MIRROR_ATTR_READING_SNAPSHOT_TIME
- CLD_SM_MIRROR_ATTR_CURRENT_MAX_DEMAND_DELIVERED_TIME
- CLD_SM_MIRROR_ATTR_CURRENT_MAX_DEMAND_RECEIVED_TIME
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_1_SUMMATION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_1_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_2_SUMMATION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_2_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_3_SUMMATION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_3_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_4_SUMMATION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_4_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_5_SUMMATION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_5_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_6_SUMMATION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_6_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_MULTIPLIER
- CLD_SM_MIRROR_ATTR_DIVISOR
- CLD_SM_MIRROR_ATTR_DEMAND_FORMATING
- CLD_SM_MIRROR_ATTR_HISTORICAL_CONSUMPTION_FORMATTING
- CLD_SM_MIRROR_ATTR_INSTANTANEOUS_DEMAND
- CLD_SM_MIRROR_ATTR_CURRENT_DAY_CONSUMPTION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_DAY_CONSUMPTION_RECEIVED
- CLD_SM_MIRROR_ATTR_PREVIOUS_DAY_CONSUMPTION_DELIVERED
- CLD_SM_MIRROR_ATTR_PREVIOUS_DAY_CONSUMPTION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_RECEIVED
- CLD_SM_MIRROR_ATTR_MAX_NUMBER_OF_PERIODS_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_DEMAND_DELIVERED
- CLD_SM_MIRROR_ATTR_DEMAND_LIMIT
- CLD_SM_MIRROR_ATTR_DEMAND_INTEGRATION_PERIOD
- CLD_SM_MIRROR_ATTR_NUMBER_OF_DEMAND_SUBINTERVALS

The Basic cluster attributes that are supported by mirroring must also be defined on the mirroring server (the same set of attributes are mirrored on all endpoints), from the following:

- CLD_BAS_MIRROR_ATTR_APPLICATION_VERSION
- CLD_BAS_MIRROR_ATTR_STACK_VERSION
- CLD_BAS_MIRROR_ATTR_HARDWARE_VERSION
- CLD_BAS_MIRROR_ATTR_MANUFACTURER_NAME

- CLD_BAS_MIRROR_ATTR_MODEL_IDENTIFIER
- CLD_BAS_MIRROR_ATTR_DATE_CODE
- CLD_BAS_MIRROR_ATTR_LOCATION_DESCRIPTION
- CLD_BAS_MIRROR_ATTR_PHYSICAL_ENVIRONMENT
- CLD_BAS_MIRROR_ATTR_DEVICE_ENABLED
- CLD_BAS_MIRROR_ATTR_ALARM_MASK
- CLD_BAS_MIRROR_ATTR_DISABLE_LOCAL_CONFIG

Get Profile If the 'Get Profile' feature is to be used (see Section 42.6), the following options must be defined in the **zcl_options.h** file.

The 'Get Profile' option must be enabled on the server and clients by including:

```
#define CLD_SM_SUPPORT_GET_PROFILE
```

Then, the following must be included on the server (only):

```
#ifdef CLD_SM_SUPPORT_GET_PROFILE
#define     CLD_SM_GETPROFILE_MAX_NO_INTERVALS     <n>
#endif
```

where <n> is the maximum number of consumption intervals to be held on the server (and therefore determines the amount of memory to be reserved for the circular buffer that is used to store the data for these consumption intervals).

Parent topic:*Simple Metering Cluster*

Part X: Commissioning Clusters This part comprises two chapters:

- *Chapter 43* details the **Commissioning** cluster
- *Chapter 44* details the **Touchlink Commissioning** cluster

Commissioning Cluster This chapter details the Commissioning cluster which is defined in the ZCL and is a optional cluster for all ZigBee devices.

The Commissioning cluster has a Cluster ID of 0x0015.

Overview The Commissioning cluster is used for commissioning the ZigBee stack on a device during network installation and defining the device behaviour with respect to the ZigBee network (it does not affect applications operating on the devices).

- The Commissioning cluster server must be implemented on a device that is to be commissioned into a network.
- The Commissioning cluster client must be implemented on a device that can initiate the commissioning of another device into a network - for example, on a commissioning tool.

This optional cluster is enabled by defining CLD_COMMISSIONING in the **zcl_options.h** file. The inclusion of the client or server software must also be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance). The compile-time options for the Commissioning cluster are fully detailed in Section 43.10.

Only server attributes are supported and all are optional - the required attributes must be enabled in the compile-time options. The information that can potentially be stored in the Commissioning cluster is organised into the following attribute sets: Start-up Parameters, Join Parameters, End Device Parameters, Concentrator Parameters.

Note: The attribute values are set by the application but the application must ensure that these values are synchronized with the settings and NIB values for the ZigBee PRO stack.

The Commissioning cluster also provides optional commands, which can be enabled in the compile-time options.

Parent topic:*Commissioning Cluster*

Commissioning Cluster structure and attributes The Commissioning cluster has only server attributes that are contained in the following tsCLD_Commissioning structure, organised as a set of structures containing the Commissioning cluster attribute sets:

```
typedef struct
{
#ifdef COMMISSIONING_SERVER
/* Start- attribute setup Parameters attribute set */
    tsCLD_StartupParameters          sStartupParameters;
/* Join Parameters attribute set */
    tsCLD_JoinParameters             sJoinParameters;
/* End Device Parameters attribute set */
    tsCLD_EndDeviceParameters        sEndDeviceParameters;
/* Concentrator Parameters attribute set */
    tsCLD_ConcentratorParameters     sConcentratorParameters;
#endif
    zuint16        u16ClusterRevision;
} tsCLD_Commissioning;
```

where:

```
sStartupParameters is a structure containing the attributes of the Start-up Parameters attribute set -
↪ this structure and the associated attributes are detailed in [Section 43.2.1](start-up_parameters_tscld_
↪startupparameters.md#id_d22b5600-7fda-403a-90c7-d433548a4983)
sJoinParameters is a structure containing the attributes of the Join Parameters attribute set -
↪ this structure and the associated attributes are detailed in [Section 43.2.2](join_parameters_tscld_
↪joinparameters.md#id_02f93815-3ce8-498e-a271-cd3675dc7015)
sEndDeviceParameters is a structure containing the attributes of the End Device Parameters attribute set -
↪ this structure and the associated attributes are detailed in [Section 43.2.3](end_device_parameters_tscld_
↪enddeviceparameters.md#id_a29fedbf-ef44-458b-88af-f134a455d4c0)
sConcentratorParameters is a structure containing the attributes of the Concentrator Parameters attribute set -
↪ this structure and the associated attributes are detailed in [Section 43.2.4](concentrator_parameters_tscld_
↪concentratorparamete.md#id_3713b9e8-5da6-4db8-b678-617069de5091)
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specifica-
tion on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster re-
vision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This at-
tribute is also described in [Section 2.4](../../ZCL_fundamentals/topics/global_attributes.md#id_8f80e769-
↪8ccd-4b4f-8609-579f4457d527).
```

Note: Memory is allocated at compile-time for all the Commissioning cluster attributes.

Start-up Parameters (tsCLD_StartupParameters) The tsCLD_StartupParameters structure below contains the attributes of the Start-up Parameters attribute set:

```
typedef struct
{
#ifdef    CLD_COMMISSIONING_ATTR_SHORT_ADDRESS
    uint16          u16ShortAddress;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_EXTENED_PAN_ID
    zieeeaddress    u64ExtPanId;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_PAN_ID
    uint16          u16PANId;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_CHANNEL_MASK
    zbmap32         u32ChannelMask;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_PROTOCOL_VERSION
    uint8           u8ProtocolVersion;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_STACK_PROFILE
    uint8           u8StackProfile;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_START_UP_CONTROL
    zenum8          e8StartUpControl;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_TC_ADDR
    zieeeaddress    u64TcAddr;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_TC_MASTER_KEY
    tsZCL_Key       sTcMasterKey;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_NWK_KEY
    tsZCL_Key       sNwkKey;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_USE_INSECURE_JOIN
    bool_t          bUseInsecureJoin;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_PRE_CONFIG_LINK_KEY
    tsZCL_Key       sPreConfigLinkKey;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_NWK_KEY_SEQ_NO
    uint8           u8NwkKeySeqNo;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_NWK_KEY_TYPE
    zenum8          e8NwkKeyType;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_NWK_MANAGER_ADDR
    uint16          u16NwkManagerAddr;
#endif
} tsCLD_StartupParameters;
```

where:

- u``16ShortAddress is the intended 16-bit network address of the device (which will be used provided that the address is not to be obtained from the parent - that is, on the Co-ordinator or on other ZigBee PRO devices for which e8StartUpControl is set to 0x00).
- u64ExtPanId is the 64-bit Extended PAN ID of the network which the device should join (the special value of 0xFFFFFFFF can be used to specify no particular network).
- u16PANId is the 16-bit PAN ID of the network which the device should join (which will be used provided that the PAN ID is not to be obtained from the parent - that is, on the Co-ordinator or on other ZigBee PRO devices for which e8StartUpControl is set to 0x00).
- u32ChannelMask is a 32-bit bitmap representing an IEEE 802.15.4 channel mask which in-

dicates the set of radio channels that the device should scan as part of the network join or formation process.

- u8ProtocolVersion is used to indicate the ZigBee protocol version that the device is to support (only needed if the device potentially supports multiple versions).
- u8StackProfile is used to indicate the stack profile to be implemented on the device - the possible values are 0x01 for ZigBee Stack profile and 0x02 for ZigBee PRO Stack profile.
- e8StartUpControl is an enumeration which is used to indicate the start-up mode of the device (e.g. device should form a network with the specified Extended PAN ID) and therefore determines how certain other attributes will be used. For further information on how this attribute is used, refer to the ZCL Specification.
- u64TcAddr is the 64-bit IEEE/MAC address of the Trust Centre node for the network with the specified Extended PAN ID (this is needed if security is to be implemented).
- sTcMasterKey is the master key to be used during key establishment with the specified Trust Centre (this is needed if security is to be implemented). The default is a 128-bit zero value indicating that the key is unspecified.
- sNwkKey is the network key to be used when communicating within the network with the specified Extended PAN ID (this is needed if security is to be implemented). The default is a 128-bit zero value indicating that the key is unspecified.
- bUseInsecureJoin is a Boolean flag which, when set to TRUE, allows an unsecured join as a fall-back (even if security is enabled).
- sPreConfigLinkKey is the pre-configured link key between the device and the Trust Centre (this is needed if security is to be implemented). The default is a 128-bit zero value indicating that the key is unspecified.
- u8NwkKeySeqNo is the 8-bit sequence number for the network key. The default value is 0x00.
- e8NwkKeyType is the type of the network key. The default value is 0x01 when u8StackProfile is 0x01 and 0x05 when u8StackProfile is 0x02.
- u16NwkManagerAddr is the 16-bit network address of the Network Manager. The default value is 0x0000, indicating that the Network Manager is the ZigBee Co-ordinator.

Parent topic: Commissioning Cluster structure and attributes

Join Parameters (tsCLD_JoinParameters) The tsCLD_JoinParameters structure below contains the attributes of the Join Parameters attribute set:

```
typedef struct
{
#ifdef    CLD_COMMISSIONING_ATTR_SCAN_ATTEMPTS
   uint8        u8ScanAttempts;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_TIME_BW_SCANS
   uint16       u16TimeBwScans;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_REJOIN_INTERVAL
   uint16       u16RejoinInterval;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_MAX_REJOIN_INTERVAL
   uint16       u16MaxRejoinInterval;
#endif
} tsCLD_JoinParameters;
```

where:

- u8ScanAttempts is the number of scan attempts to make before selecting a parent to join. The default value is 0x05.
- u16TimeBwScans is the time-interval, in milliseconds, between consecutive scan attempts. The default value is 0x64.
- u16RejoinInterval is the time-interval, in seconds, between consecutive attempts to rejoin the network for an End Device which has lost its network connection. The default value is 0x3C.
- u16MaxRejoinInterval is an upper limit, in seconds, on the value of the u16RejoinInterval attribute. The default value is 0x0E10.

Parent topic: Commissioning Cluster structure and attributes

End Device Parameters (tsCLD_EndDeviceParameters) The tsCLD_EndDeviceParameters structure below contains the attributes of the End Device Parameters attribute set:

```
typedef struct
{
#ifdef    CLD_COMMISSIONING_ATTR_INDIRECT_POLL_RATE
    uint16        u16IndirectPollRate;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_PARENT_RETRY_THRSHLD
    uint8         u8ParentRetryThreshold;
#endif
} tsCLD_EndDeviceParameters;
```

where:

- u16IndirectPollRate is the time-interval, in milliseconds, between consecutive polls from an End Device which polls its parent while awake (an End Device with a receiver that is inactive while sleeping).
- u8ParentRetryThreshold is the number of times that an End Device should attempt to re-contact its parent before initiating the rejoin process.

Parent topic: Commissioning Cluster structure and attributes

Concentrator Parameters (tsCLD_ConcentratorParameters) The sCLD_ConcentratorParameters structure below contains the attributes of the Concentrator Parameters attribute set:

```
typedef struct
{
#ifdef    CLD_COMMISSIONING_ATTR_CONCENTRATOR_FLAG
    bool_t        bConcentratorFlag;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_CONCENTRATOR_RADIUS
    uint8         u8ConcentratorRadius;
#endif
#ifdef    CLD_COMMISSIONING_ATTR_CONCENTRATOR_DISCVRY_TIME
    uint8         u8ConcentratorDiscoveryTime;
#endif
} tsCLD_ConcentratorParameters;
```

where:

- bConcentratorFlag is a Boolean flag which, when set to TRUE, enables the device as a concentrator for many-to-one routing. The default value is FALSE.

- u8ConcentratorRadius is the hop-count radius for concentrator route discoveries. The default value is 0x0F.
- u8ConcentratorDiscoveryTime is the time-interval, in seconds, between consecutive discoveries of inbound routes initiated by the concentrator. The default value is 0x0000, indicating that this time-interval is unknown and the discoveries must be triggered by the application.

Parent topic: Commissioning Cluster structure and attributes

Parent topic:*Commissioning Cluster*

Attribute Settings The Commissioning cluster structure contains only optional attributes. Each attribute is enabled/disabled through a corresponding macro defined in the **zcl_options.h** file (see Section 43.10) - for example, u16ShortAddress is enabled/disabled through the macro CLD_COMM_ATTR_SHORT_ADDRESS.

The function **eCLD_CommissioningSetAttribute()** can be used on the cluster server to write values to any one of the four attribute sets of the Commissioning cluster.

Parent topic:*Commissioning Cluster*

Initialisation The function **eCLD_CommissioningClusterCreateCommissioning()** is used to create an instance of the Commissioning cluster. The function is generally called by the initialization function for the host device.

Parent topic:*Commissioning Cluster*

Commissioning Commands A number of commissioning commands are provided to allow a Commissioning cluster client to remotely request actions relating to the Start-up Parameters attribute set (see Section 43.2.1) on a cluster server. This includes initiating a device restart from the current Start-up Parameter values, as well as the management of these attributes.

Device Start-up The 'current' set of Start-up Parameter values on a cluster server are those used in the start-up procedure, which can be remotely initiated from a cluster client using the function **eCLD_CommissioningCommandRestartDeviceSend()**. This function sends a Restart Device command to the remote device hosting the cluster server. This command provides a number of options concerning the timing of the restart:

- **Without delay:** The start-up procedure is invoked as soon as the command is received. This option requires both the delay and jitter to be specified as zero.
- **With delay:**The start-up procedure is invoked after a specified delay (in seconds). If no delay is required, the delay period must be specified as zero.
- **With delay and jitter:** The start-up procedure is invoked after a specified delay (in seconds) with a random jitter period added. It is necessary to indicate a maximum jitter period but the actual period will be randomly generated. If no jitter is required, the maximum jitter period must be specified as zero.

Note: If only jitter is required, the delay period must be specified as zero and the maximum jitter period must be non-zero.

In all of the above cases, it is possible to configure the start-up procedure to begin either without any further delay or at a 'convenient' moment (when there are no pending actions that should be completed before the restart).

The above options are configured in the command payload (see Section 43.9.2).

The cluster server will send a Restart Device Response to the requesting client before invoking the start-up procedure or starting the countdown (for the delay).

Parent topic: Commissioning Commands

Stored Start-up Parameters In addition to the 'current' set of values for the Start-up Parameters attribute set, the cluster server can store other sets of values for these attributes in non-volatile memory. Each stored set of Start-up Parameter values is assigned a unique index number. At any time, a particular stored set of values can be retrieved and loaded to become the current set. Functions are provided for managing the saved sets of Start-up Parameter values.

Saving Start-up Parameters In order to save a set of Start-up Parameter values, it is first necessary to set them as the current attribute values - this must be done locally by the application on the device hosting the server, possibly using the function **eCLD_CommissioningSetAttribute()**.

The application on a device hosting a cluster client can send a Save Start-up Parameters command to the cluster server in order to request that the current set of Start-up Parameter values is saved to non-volatile memory. This can be done by calling **eCLD_CommissioningCommandSaveStartupParamsSend()** or, alternatively, **eCLD_CommissioningCommandModifyStartupParamsSend()**. The index number of the saved record must be specified in the request. If this number has already been used, the existing stored values will be over-written with the new values.

It is the responsibility of the user application on the device hosting the server to perform the save. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located. The server will automatically send a Save Start-up Parameters Response to the requesting client.

Parent topic: Stored Start-up Parameters

Retrieving Stored Start-up Parameters A set of Start-up Parameter values that have been stored by in non-volatile memory (as described in Section 43.5.2.1) can be retrieved and loaded as the current set of values. The required stored set of values is specified using its unique index number.

The application on a device hosting a cluster client can send a Restore Start-up Parameters command to the cluster server in order to request that the specified set of Start-up Parameter values is loaded from non-volatile memory. This can be done by calling **eCLD_CommissioningCommandRestoreStartupParamsSend()** or, alternatively, **eCLD_CommissioningCommandModifyStartupParamsSend()**. The index number of the relevant set must be specified in the request.

It is the responsibility of the user application on the device hosting the server to retrieve the relevant set of values and load them as the current values. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located. The server will automatically send a Restore Start-up Parameters Response to the requesting client.

A device restart is required in order to implement the loaded values, as described in Section 43.5.1.

Parent topic: Stored Start-up Parameters

Parent topic: Commissioning Commands

Reset Start-up Parameters to Default Values A set of Start-up Parameters on the cluster server can be reset to their default values.

The application on a device hosting a cluster client can send a Reset Start-up Parameters command to the cluster server in order to request that the Start-up Parameters are reset to their default values. This can be done by calling **eCLD_CommissioningCommandResetStartupParamsSend()** or, alternatively,

eCLD_CommissioningCommandModifyStartupParamsSend (). Options are available concerning the set(s) of Start-up Parameters to reset - any combination of the following can be performed:

- Reset the current set of Start-up Parameters
- Reset all stored sets of Start-up Parameters or the stored set with given index
- Erase the stored set of Start-up Parameters with given index

The required options must be specified in the request. The option to erase a stored set of Start-up Parameters allows storage space to be freed up.

It is the responsibility of the user application on the device hosting the server to reset the relevant set(s) of values. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located. The server will automatically send a Reset Start-up Parameters Response to the requesting client.

A device restart is required in order to implement the reset (current) values, as described in Section 43.5.1.

Parent topic: Commissioning Commands

Parent topic:*Commissioning Cluster*

Commissioning Events The Commissioning cluster has its own events that are handled through the callback mechanism outlined in *Chapter 3*. If a device uses this cluster then application-specific Commissioning event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function will then be invoked when a Commissioning event occurs and needs the attention of the application.

For a Commissioning event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_CommissioningCallBackMessage structure (fully detailed in Section43.9.6).

```
typedef struct
{
    uint8                    u8CommandId;
    union
    {
        tsCLD_Commissioning_RestartDevicePayload
                        *psRestartDevicePayload;
        tsCLD_Commissioning_ModifyStartupParametersPayload
                        *psModifyStartupParamsPayload;
    } uReqMessage;
    union
    {
        tsCLD_Commissioning_ResponsePayload
                        *psCommissioningResponsePayload;
    } uRespMessage;
} tsCLD_CommissioningCallBackMessage;
```

When a Commissioning event occurs, one of a number of command types could have been received. The relevant command type is specified through the u8CommandId field of the tsCLD_CommissioningCallBackMessage structure. The possible command types are detailed below.

The table below details the command types that can be received by the cluster server.

u8CommandId Enumeration	Description
E_CLD_COMMISSIONING_CMD_RESTART_DEVI	A Restart Device command has been received
E_CLD_COMMISSIONING_CMD_SAVE_STARTUP	A Save Start-up Parameters command has been received
E_CLD_COMMISSIONING_CMD_RESTORE_STAR	A Restore Start-up Parameters command has been received
E_CLD_COMMISSIONING_CMD_RESET_STARTU	A Reset Start-up Parameters command has been received

The table below details the command types that can be received by the cluster client.

u8CommandId Enumeration	Description
E_CLD_COMMISSIONING_CMD_RESTART_DEVI(	A Restart Device response has been received
E_CLD_COMMISSIONING_CMD_SAVE_STARTUP_	A Save Start-up Parameters response has been received
E_CLD_COMMISSIONING_CMD_RESTORE_STAR'	A Restore Start-up Parameters response has been received
E_CLD_COMMISSIONING_CMD_RESET_STARTU:	A Reset Start-up Parameters response has been received

Parent topic:*Commissioning Cluster*

Functions The following Commissioning cluster function is provided:

1. eCLD_CommissioningClusterCreateCommissioning
2. eCLD_CommissioningCommandRestartDeviceSend
3. eCLD_CommissioningCommandSaveStartupParamsSend
4. eCLD_CommissioningCommandRestoreStartupParamsSend
5. eCLD_CommissioningCommandResetStartupParamsSend
6. eCLD_CommissioningCommandModifyStartupParamsSend
7. eCLD_CommissioningSetAttribute

eCLD_CommissioningClusterCreateCommissioning

```
teZCL_Status eCLD_CommissioningClusterCreateCommissioning(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits,
    tsCLD_CommissioningCustomDataStructure
    *psCustomDataStructure);
```

Description This function creates an instance of the Commissioning cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Commissioning cluster instance on the endpoint, but instances of

other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to *Appendix D.*

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in the *ZigBee Devices User Guide*

Note: *(JNUG3131)*.

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length is automatically adjusted by the compiler using the following declaration:

```
uint8 au8CommissioningAttributeControlBits
[(sizeof(asCLD_CommissioningClusterAttributeDefinitions) / sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Commissioning cluster. This parameter can refer to a pre-filled structure called sCLD_Commissioning which is provided in the **Commissioning.h** file.
- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_Commissioning which defines the attributes of Commissioning cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.
- *psCustomDataStructure* Pointer to a structure containing the storage for internal functions of the cluster (see Section 43.9.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_CommissioningCommandRestartDeviceSend

```
teZCL_Status eCLD_CommissioningCommandRestartDeviceSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber, tsCLD_Commissioning_RestartDevicePayload *psPayload);
```

Description This function can be used on a Commissioning cluster client to send a Restart Device command to a cluster server on a remote device. This command is used to run the start-up procedure with a new set of values for the Start-up Parameters attributes (these values must already be installed). The new values may be implemented immediately or after a specified delay with an optional jitter.

When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located (see Section 43.6). Before running the start-up procedure or starting the countdown (for the delay), the server will send a Restart Device Response to the requesting client, where a ZCL custom event will be generated.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to a structure containing the payload for this message (see Section 43.9.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_CommissioningCommandSaveStartupParamsSend

```
teZCL_Status eCLD_CommissioningCommandSaveStartupParamsSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
```

(continues on next page)

(continued from previous page)

```
uint8 *pu8TransactionSequenceNumber
tsCLD_Commissioning_ModifyStartupParametersPayload *psPayload);
```

Description This function can be used on a Commissioning cluster client to send a Save Start-up Parameters command to a cluster server on a remote device. This command instructs the server to locally save a set of values for the attributes of the Start-up Parameters attribute set. A device can store different sets of start-up parameters (in non-volatile memory), with each set being referenced using an index number. This index number must be specified and if a set has already been stored with the same index number then the stored values will be over-written with the new values.

It is the responsibility of the user application on the device hosting the server to implement the command. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located (see Section 43.6). The server will automatically send a Save Start-up Parameters Response to the client, where a ZCL custom event will be generated.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to a structure containing the payload for this message (see Section 43.9.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_CommissioningCommandRestoreStartupParamsSend

```
teZCL_Status eCLD_CommissioningCommandRestoreStartupParamsSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber, tsCLD_Commissioning_*psPayload)
```

Description This function can be used on a Commissioning cluster client to send a Restore Start-up Parameters command to a cluster server on a remote device. This command instructs the server to load a saved set of values for the attributes of the Start-up Parameters attribute set. The index of the required set of Start-up Parameters must be specified in the command payload. Note that the command does not instruct the server to implement the loaded values using the start-up procedure - a Restart Device command is required to do this.

It is the responsibility of the user application on the device hosting the server to implement the command. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located (see Section 43.6). The server will automatically send a Restore Start-up Parameters Response to the client, where a ZCL custom event will be generated.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to a structure containing the payload for this message (see Section 43.9.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_CommissioningCommandResetStartupParamsSend

```
teZCL_Status eCLD_CommissioningCommandResetStartupParamsSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber, tsCLD_*psPayload);
```

Description This function can be used on a Commissioning cluster client to send a Reset Start-up Parameters command to a cluster server on a remote device. This command instructs the server to set the current Start-up Parameters to their default values. It is also possible to set one or all of any saved sets of Start-up Parameters to the defaults. The command can also be used to delete a specified set of saved Start-up Parameters.

It is the responsibility of the user application on the device hosting the server to implement the command. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located (see Section 43.6). The server will automatically send a Reset Start-up Parameters Response to the client, where a ZCL custom event will be generated.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to a structure containing the payload for this message (see Section 43.9.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_CommissioningCommandModifyStartupParamsSend

```
teZCL_Status eCLD_CommissioningCommandModifyStartupParamsSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_Commissioning_ModifyStartupParametersPayload *psPayload,
    teCLD_Commissioning_Command eCLD_Commissioning_Command);
```

Description This function can be used on a Commissioning cluster client to send a command to modify a set of values for the Start-up Parameters attributes in the cluster server on a remote device. One of four commands can be specified and sent, as listed and described in the table below:

Command	Description
Restart Device	Used to run the start-up procedure with the current set of values for the Start-up Parameters attributes, as described for the function **eCLD_CommissioningCommandRestartDeviceSend()**. These values may have been loaded using the Restore Start-up Parameters or Reset Start-up Parameters command.
Save Start-up Parameters	Used to save the current set of Start-up Parameter values with the specified index, as described for the function **eCLD_CommissioningCommandSaveStartupParamsSend()**.
Restore Start-up Parameters	Used to load the saved set of Start-up Parameter values with the specified index, such that these values become the current Start-up Parameter val-ues, as described for the function **eCLD_CommissioningCommandRe-storeStartupParamsSend()**. Note that these values are not implemented, which requires a Restart Device command.
Reset Start-up Parameters	Used to reset the current Start-up Parameters to their defaults. One or all of any stored sets of Start-up Parameter values can also be reset to the defaults, as described for the function **eCLD_CommissioningComman-dResetStartupParamsSend()**. The command can also be used to delete a particular set of stored Start-up Parameters.

It is the responsibility of the user application on the device hosting the server to implement the command. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located (see Section 43.6). The server will automatically send a response for the relevant command to the client, where a ZCL custom event will be generated.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values

- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to a structure containing the payload for this message (see Section 43.9.3)
- *eCLD_Commissioning_Command* Type of command to send, one of:
- E_CLD_COMMISSIONING_CMD_RESTART_DEVICE
- E_CLD_COMMISSIONING_CMD_SAVE_STARTUP_PARAMS
- E_CLD_COMMISSIONING_CMD_RESTORE_STARTUP_PARAMS
- E_CLD_COMMISSIONING_CMD_RESET_STARTUP_PARAMS

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_CommissioningSetAttribute

```
teZCL_Status eCLD_CommissioningSetAttribute(
    uint8 u8SourceEndPointId,
    teCLD_Commissioning_AttributeSet eAttributeSet,
    void *vptrAttributeSetStructure);
```

Description { This function can be used on a Commissioning cluster server to write values to a particular attribute set of the Commissioning cluster.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to issue the request.
- *eAttributeSet* Enumeration indicating attribute set to write to, one of:
- E_CLD_COMMISSIONING_ATTR_SET_STARTUP_PARAMS
- E_CLD_COMMISSIONING_ATTR_SET_JOIN_PARAMS
- E_CLD_COMMISSIONING_ATTR_SET_ENDDEVICE_PARAMS
- E_CLD_COMMISSIONING_ATTR_SET_CONCENTRATOR_PARAMS
- *vptrAttributeSetStructure* Pointer to a structure containing the new values for the attribute set - the relevant structures are detailed in Section 43.2

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic:*Commissioning Cluster*

Enumerations

teCLD_Commissioning_AttributeID The following structure contains the enumerations used to identify the attributes of the Commissioning cluster.

```
typedef enum
{
    E_CLD_COMMISSIONING_ATTR_ID_SHORT_ADDRESS          = 0x0000,
    E_CLD_COMMISSIONING_ATTR_ID_EXT_PANID,
    E_CLD_COMMISSIONING_ATTR_ID_PANID,
    E_CLD_COMMISSIONING_ATTR_ID_CHANNEL_MASK,
    E_CLD_COMMISSIONING_ATTR_ID_PROTOCOL_VERSION,
    E_CLD_COMMISSIONING_ATTR_ID_STACK_PROFILE,
    E_CLD_COMMISSIONING_ATTR_ID_STARTUP_CONTROl,
    E_CLD_COMMISSIONING_ATTR_ID_TC_ADDR                = 0x0010,
    E_CLD_COMMISSIONING_ATTR_ID_TC_MASTER_KEY,
    E_CLD_COMMISSIONING_ATTR_ID_NETWORK_KEY,
    E_CLD_COMMISSIONING_ATTR_ID_USE_INSECURE_JOIN,
    E_CLD_COMMISSIONING_ATTR_ID_PRECONFIG_LINK_KEY,
    E_CLD_COMMISSIONING_ATTR_ID_NWK_KEY_SEQ_NO,
    E_CLD_COMMISSIONING_ATTR_ID_NWK_KEY_TYPE,
    E_CLD_COMMISSIONING_ATTR_ID_NWK_MANAGER_ADDR,
    E_CLD_COMMISSIONING_ATTR_ID_SCAN_ATTEMPTS          = 0x0020,
    E_CLD_COMMISSIONING_ATTR_ID_TIME_BW_SCANS,
    E_CLD_COMMISSIONING_ATTR_ID_REJOIN_INTERVAL,
    E_CLD_COMMISSIONING_ATTR_ID_MAX_REJOIN_INTERVAL,
    E_CLD_COMMISSIONING_ATTR_ID_INDIRECT_POLL_RATE     = 0x0030,
    E_CLD_COMMISSIONING_ATTR_ID_PARENT_RETRY_THRSHOLD,
    E_CLD_COMMISSIONING_ATTR_ID_CONCENTRATOR_FLAG      = 0x0040,
    E_CLD_COMMISSIONING_ATTR_ID_CONCENTRATOR_RADIUS,
    E_CLD_COMMISSIONING_ATTR_ID_CONCENTRATOR_DISCVRY_TIME,
} teCLD_Commissioning_AttributeID;;
```

Parent topic: Enumerations

teCLD_Commissioning_AttributeSet The following structure contains the enumerations used to identify the attribute sets of the Commissioning cluster.

```
typedef enum
{
    E_CLD_COMMISSIONING_ATTR_SET_STARTUP_PARAMS      = 0x00,
    E_CLD_COMMISSIONING_ATTR_SET_JOIN_PARAMS,
    E_CLD_COMMISSIONING_ATTR_SET_ENDDEVICE_PARAMS,
    E_CLD_COMMISSIONING_ATTR_SET_CONCENTRATOR_PARAMS
} teCLD_Commissioning_AttributeSet;
```

Parent topic: Enumerations

teCLD_Commissioning_Command The following structure contains the enumerations used to identify commands of the Commissioning cluster (the same enumerations are used for requests and their corresponding responses).

```
typedef enum
{
    E_CLD_COMMISSIONING_CMD_RESTART_DEVICE              = 0x00,
    E_CLD_COMMISSIONING_CMD_SAVE_STARTUP_PARAMS,
    E_CLD_COMMISSIONING_CMD_RESTORE_STARTUP_PARAMS,
    E_CLD_COMMISSIONING_CMD_RESET_STARTUP_PARAMS
} teCLD_Commissioning_Command;
```

The above enumerations are described in the table below:

Enumeration	Command
E_CLD_COMMISSIONING_CMD_RESTART_DEVICE	Restart Device request or response
E_CLD_COMMISSIONING_CMD_SAVE_STARTUP_PAI	Save Start-up Parameters request or response
E_CLD_COMMISSIONING_CMD_RESTORE_STARTUP_	Restore Start-up Parameters request or response
E_CLD_COMMISSIONING_CMD_RESET_STARTUP_PA	Reset Start-up Parameters request or response

Parent topic: Enumerations

Parent topic:*Commissioning Cluster*

Structures

Attribute Set Structures The following structures contain the Commissioning cluster attribute sets and are detailed in the referenced sections:

- tsCLD_StartupParameters - see Section 43.2.1
- tsCLD_JoinParameters - see Section 43.2.2
- tsCLD_EndDeviceParameters - see Section 43.2.3
- tsCLD_ConcentratorParameters - see Section 43.2.4

Parent topic: Structures

tsCLD_Commissioning_RestartDevicePayload The following structure contains the payload of a Restart Device command.

```
typedef struct
{
    zbmap8                  u8Options;
    uint8                   u8Delay;
    uint8                   u8Jitter;
} tsCLD_Commissioning_RestartDevicePayload;
```

where:

```
u8Options is a 8-bit bitmap specifying the required start-up options:
```

Bits	Option	Description
0-2	Start-up Mode	Determines the starting state of the device restart:

- 0b000: Restart with current Start-up Parameter values
- 0b001: Restart from existing stack state

All other values are reserved.

| |3|Immediate|Determines how quickly the start-up procedure will begin follow-ing receipt of the command or the specified delay/jitter:

- 1: Immediately
- 0: At a convenient moment (e.g. following any pending actions)

| |4-7|-|Reserved|

```
u8Delay specifies the time-delay, in seconds, before the start-up procedure should be executed.
u8Jitter is a value which determines the possible range of values of the jitter that is added to the delay u8Delay -
↪ the jitter, in milliseconds, is a random value in the range 0 to u8Jitter x 80.
```

Parent topic: Structures

tsCLD_Commissioning_ModifyStartupParametersPayload The following structure contains the payload of the following commands: Save Start-up Parameters, Restore Start-up Parameters and Reset Start-up Parameters.

```
typedef struct
{
    zbmap8                  u8Options;
    uint8                   u8Index;
} tsCLD_Commissioning_ModifyStartupParametersPayload;
```

where:

```
u8Options is an 8-bit bitmap specifying the required reset options for the Reset Start-
↪up Parameters command (it is not used by the other commands):
```

Bits	Option	Description
0	Reset Current	Determines whether the current Start-up Parameters will be reset to their default values:

- 1: Reset to default values
- 0: Do not reset (remain unchanged)

| |1|Reset All|Determines whether all stored Start-up Parameter sets will be reset to their default values:

- 1: Reset all stored Start-up Parameter sets
- 0: Reset the stored Start-up Parameter set with specified index

| |2|Erase Index|Determines whether the stored Start-up Parameter set with speci-fied index will be erased:

- 1: Erase Start-up Parameter set with specified index
- 0: Do not erase Start-up Parameter set with specified index

| |3-7|-|Reserved|

```
u8Index is the index of the saved Start-
↪up Parameter set to which actions specified in u8Options relate (this index is ignored if no actions are specified).
```

Parent topic: Structures

tsCLD_Commissioning_ResponsePayload The following structure contains the payload of the responses to the following commands: Save Start-up Parameters, Restore Start-up Parameters and Reset Start-up Parameters.

```
typedef struct
{
    zenum8                  u8Status;
} tsCLD_Commissioning_ResponsePayload;
```

where u8Status contains one of the ZCL command status codes listed and described in Section 7.1.4.

Parent topic: Structures

tsCLD_CommissioningCustomDataStructure The Commissioning cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress               sReceiveEventAddress;
    tsZCL_CallBackEvent                     sCustomCallBackEvent;
    tsCLD_CommissioningCallBackMessage      sCallBackMessage;
} tsCLD_CommissioningCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

tsCLD_CommissioningCallBackMessage For a Commissioning event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_CommissioningCallBackMessage structure:

```
typedef struct
{
    uint8                       u8CommandId;
    union
    {
        tsCLD_Commissioning_RestartDevicePayload
                            *psRestartDevicePayload;
        tsCLD_Commissioning_ModifyStartupParametersPayload
                            *psModifyStartupParamsPayload;
    } uReqMessage;
    union
```

(continues on next page)

(continued from previous page)

```
    {
        tsCLD_Commissioning_ResponsePayload
                        *psCommissioningResponsePayload;
    } uRespMessage;
} tsCLD_CommissioningCallBackMessage;
```

where:

- u8CommandId indicates the type of Commissioning command that has been received by a cluster server or client (the same enumerations are used for requests on the server and responses on the client), one of:
  - E_CLD_COMMISSIONING_CMD_RESTART_DEVICE
  - E_CLD_COMMISSIONING_CMD_SAVE_STARTUP_PARAMS
  - E_CLD_COMMISSIONING_CMD_RESTORE_STARTUP_PARAMS
  - E_CLD_COMMISSIONING_CMD_RESET_STARTUP_PARAMS
- uReqMessage is a union containing the payload of a request command in the following form:
  - psRestartDevicePayload is a pointer to a structure containing the Restart Device command payload - see Section 43.9.2
  - psModifyStartupParamsPayload is a pointer to a structure containing the (common) payload for the Save Start-up Parameters, Restore Start-up Parameters and Reset Start-up Parameters commands - see Section43.9.3
- uRespMessage is a union containing the payload of a response command in the following form:
  - psCommissioningResponsePayload is a pointer to a structure containing the (common) payload for the Save Start-up Parameters, Restore Start-up Parameters and Reset Start-up Parameters responses - see Section 43.9.4

For further information on Commissioning cluster events, refer to Section 43.6.

Parent topic: Structures

Parent topic:*Commissioning Cluster*

Compile-time options To enable the Commissioning cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_COMMISSIONING
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define COMMISSIONING_CLIENT
#define COMMISSIONING_SERVER
```

Optional Attributes The Commissioning cluster contains attributes that may be optionally enabled at compile-time by adding some or all of the following lines to the **zcl_options.h** file (see Section 43.2 and Section 43.3):

```
#define CLD_COMM_ATTR_SHORT_ADDRESS
#define CLD_COMM_ATTR_EXTENED_PAN_ID
#define CLD_COMM_ATTR_PAN_ID
#define CLD_COMM_ATTR_CHANNEL_MASK
#define CLD_COMM_ATTR_PROTOCOL_VERSION
#define CLD_COMM_ATTR_STACK_PROFILE
#define CLD_COMM_ATTR_START_UP_CONTROL
    #define CLD_COMM_ATTR_TC_ADDR
#define CLD_COMM_ATTR_TC_MASTER_KEY
    #define CLD_COMM_ATTR_NWK_KEY
#define CLD_COMM_ATTR_USE_INSECURE_JOIN
    #define CLD_COMM_ATTR_PRE_CONFIG_LINK_KEY
#define CLD_COMM_ATTR_NWK_KEY_SEQ_NO
    #define CLD_COMM_ATTR_NWK_KEY_TYPE
#define CLD_COMM_ATTR_NWK_MANAGER_ADDR
#define CLD_COMM_ATTR_SCAN_ATTEMPTS
#define CLD_COMM_ATTR_TIME_BW_SCANS
#define CLD_COMM_ATTR_REJOIN_INTERVAL
#define CLD_COMM_ATTR_MAX_REJOIN_INTERVAL
    #define CLD_COMM_ATTR_INDIRECT_POLL_RATE
#define CLD_COMM_ATTR_PARENT_RETRY_THRSHLD
#define CLD_COMM_ATTR_CONCENTRATOR_FLAG
#define CLD_COMM_ATTR_CONCENTRATOR_RADIUS
#define CLD_COMM_ATTR_CONCENTRATOR_DISCVRY_TIME
```

Optional Commands The Commissioning cluster contains commands that may be optionally enabled at compile-time by adding some or all of the following lines to the **zcl_options.h** file.

To enable the Save Start-up Parameters command, add the following line:

```
#define CLD_COMMISSIONING_CMD_SAVE_STARTUP_PARAMS
```

To enable the Restore Start-up Parameters command, add the following line:

```
#define CLD_COMMISSIONING_CMD_RESTORE_STARTUP_PARAMS
```

Parent topic:*Commissioning Cluster*

Touchlink Commissioning Cluster This chapter describes the Touchlink Commissioning cluster, which can be used when forming a network or adding a new node to an existing network.

The Touchlink Commissioning cluster has a Cluster ID of 0x1000.

Overview The Touchlink Commissioning cluster is associated with a node as a whole, rather than with individual ZigBee devices on the node. It must be used on nodes that incorporate one or more of the ZigBee devices indicated in *Table 83* below, which shows the supported devices when the Touchlink Commissioning cluster acts as a client, server and combined client/server.

Client	Client/Server	Server
Colour Controller	Colour Controller	Colour Controller
Colour Scene Controller	Colour Scene Controller	Colour Scene Controller
Non-Colour Controller	Non-Colour Controller	Non-Colour Controller
Non-Colour Scene Controller	Non-Colour Scene Controller	Non-Colour Scene Controller
Control Bridge	Control Bridge	Control Bridge
On/Off Sensor	On/Off Sensor	On/Off Sensor
	On/Off Light	On/Off Light
	On/Off Plug-in Unit	On/Off Plug-in Unit
	Dimmable Light	Dimmable Light
	Dimmable Plug-in Unit	Dimmable Plug-in Unit
	Colour Light	Colour Light
	Extended Colour Light	Extended Colour Light
	Colour Temperature Light	Colour Temperature Light

This cluster supports two sets of functionality, corresponding to two distinct commands sets:

- Touchlink
- Commissioning Utility

Functions are provided for implementing both sets of commands. These functions are referenced in Section 44.4 and Section 44.5, and detailed in Section 44.7.

The Commissioning Utility functionality is not required on Lighting devices.

For the compile-time options for enabling the Touchlink Commissioning cluster for Touchlink and the Commissioning Utility, refer to Section 44.10.

Parent topic:*Touchlink Commissioning Cluster*

Cluster structure and attributes This cluster has no attributes, as a server or a client. Therefore, the cluster structure tsCLD_ZllCommission is referred to using a null pointer.

Parent topic:*Touchlink Commissioning Cluster*

Commissioning operations Commissioning involves forming a network or adding a new node to an existing network. A node from which commissioning can be initiated is referred to as an 'initiator' - this may be a remote control unit, but could also be a lamp.

- An 'initiator' node must support the Touchlink Commissioning cluster as a client.
- A node to be added to the network must support the Touchlink Commissioning cluster as a server (or as both a server and client).

Note that commissioning a new network involves adding at least one node to the new network (as well as the initiator).

Commissioning may involve two stages, depending on the type of node added to the network by the initiator:

1. The node is added to the network using the Touchlink commands of the Touchlink Commissioning cluster. In practice for the user, this typically involves bringing the initiator node physically close to the target node and pressing a button.

2. If the initiator node and the new node will both be used to control lights in the network, the new node must learn certain information (such as controlled endpoints and configured groups) from the initiator. This exchange of information uses the Commissioning Utility commands of the Touchlink Commissioning cluster.

Note: **Note:**The Touchlink Commissioning cluster instance for Touchlink must reside on its own endpoint on a node. Therefore, a Touchlink commissioning application must be provided which is distinct from the main application. However, the cluster instance for the Commissioning Utility can reside on the same endpoint as the main application (and be used in this application).

Commissioning using the supplied functions for Touchlink and the Commissioning Utilty is described in Section 44.4 and Section 44.5.

Parent topic:*Touchlink Commissioning Cluster*

Using Touchlink Touchlink is used for the basic commissioning of a new network or adding a new node to an existing network. A dedicated Touchlink application (which is distinct from the main application on the node) must reside on its own endpoint. This requires:

- a Touchlink Commissioning cluster instance as a client to be created on the endpoint on the initiator node.
- a Touchlink Commissioning cluster instance as a server to be created on the endpoint on the target node.

The initiator node also requires a Touchlink Commissioning cluster instance as a server (on the same endpoint), since the node also needs the capability to join an existing network.

An endpoint is registered for Touchlink (on both nodes) using the function **eZLL_RegisterCommissionEndPoint()**. This function also creates a Touchlink Commissioning cluster instance of the type (server, client or both) determined by the compile-time options in the header file **zcl_options.h** (see Section 44.10).

The initiator must then send a sequence of request commands to the target node. The Touchlink request command set is summarized in Table 1. Touchlink functions for issuing these commands are provided and are detailed in Section 44.7.1.

Using Touchlink

Command	Identifier	Description
Scan Request *	0x00	Requests other devices (potential nodes) in the local neighbourhood to respond. A scan request is first performed on channel 11, up to five times until a response is received. If no response is received, a scan request is then performed once on each of channels 15, 20 and 25, and then the remaining channels (12, 13, 14, 16, etc) until a response is detected.
Device Information Request *	0x02	Requests information about the devices on a remote node
Identify Request	0x06	Requests a remote node to physically identify itself (for example, visually by flashing an LED)
Reset To Factory New Request	0x07	Requests a factory reset of a remote node
Network Start Request *	0x10	Requests a new network to be created comprising the initiator and a detected Router
Network Join Router Request *	0x12	Requests a Router to join the network
Network Join End Device Request *	0x14	Requests an End Device to join the network
Network Update Request *	0x16	Requests an update of the network settings on a remote node (if the supplied Network Update Identifier is more recent than the one on the node)

* These commands have corresponding responses.

All Touchlink commands are sent as inter-PAN messages.

Use of the above commands and associated functions is described in the sub-sections below.

Creating a network A network is formed from an initiator node and a Router node (usually the initiator is an End Device and will have no routing capability in the network). The Touchlink network creation process is described below and is illustrated in the figure below. (also refer to

the command list in the table in the Section Using Touchlink).

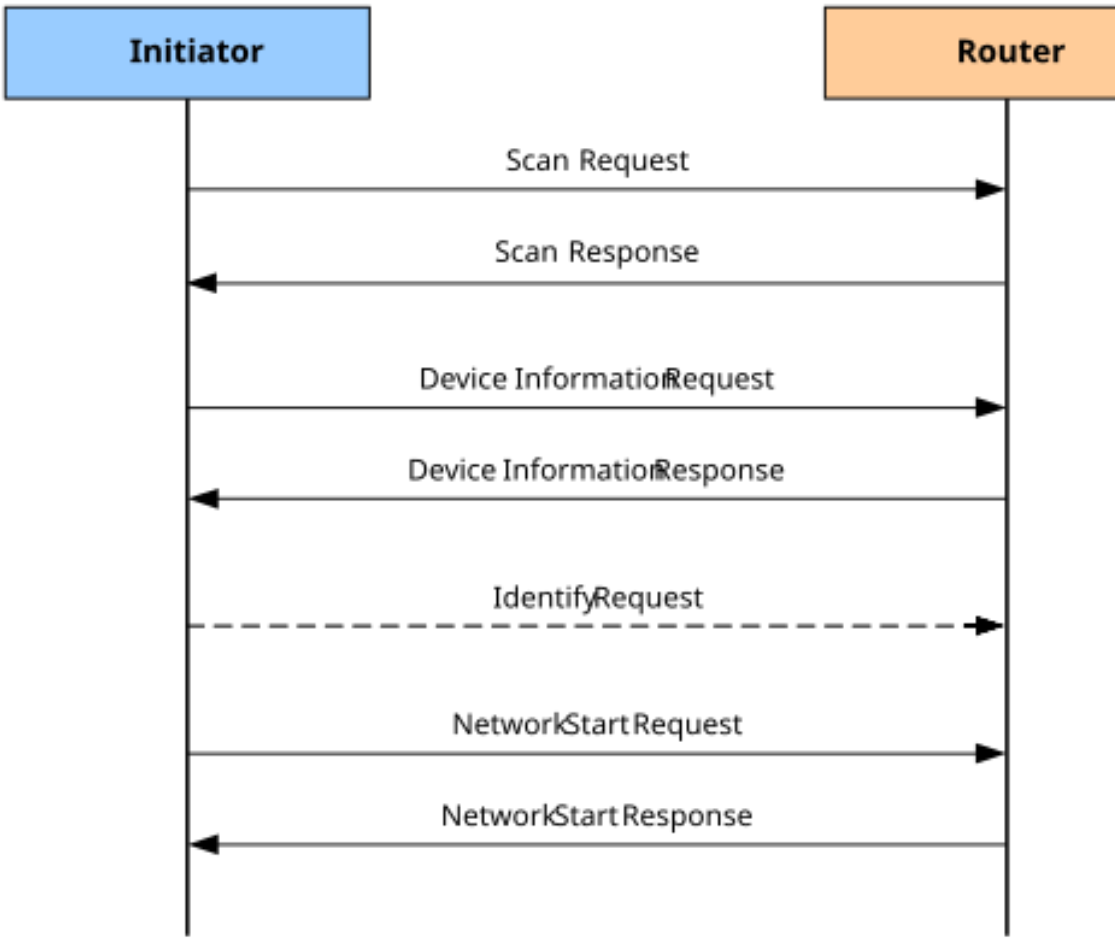
Note: Received Touchlink requests and responses are handled as ZigBee PRO events. The event handling is not detailed below but is outlined in Section 44.6.

1. **Scan Request:** The initiator sends a Scan Request to nodes in its vicinity. The required function is:
   eCLD_ZllCommissionCommandScanReqCommandSend\(\)
2. **Scan Response:** A receiving node replies to the Scan Request by sending a Scan Response, which includes the device type of the responding node (e.g. Router). The required function is: eCLD_ZllCommissionCommandScanRspCommandSend\(\)
3. **Device Information Request:** The initiator sends a Device Information Request to the detected Routers that are of interest. The required function is: eCLD_ZllCommissionCommandDeviceInfoReqCommandSend\(\)
4. **Device Information Response:** A receiving Router replies to the Device Information Request by sending a Device Information Response. The required function is:
   eCLD_ZllCommissionCommandDeviceInfoRspCommandSend\(\)
5. **Identify Request (Optional):** The initiator may send an Identify Request to the node which has been chosen as the first Router of the new network, in order to confirm that the correct physical node is being commissioned. The required function is:

   eCLD_ZllCommissionCommandDeviceIdentifyReqCommandSend\(\)
6. **Network Start Request:** The intiator sends a Network Start Request to the chosen Router in order to create and start the network. The required function is:
   eCLD_ZllCommissionCommandNetworkStartReqCommandSend\(\)
7. **Network Start Response:** The Router replies to the Network Start Request by sending a Network Start Response. The required function is: eCLD_ZllCommissionCommandNetworkStartRspCommandSend\(\)

Once the Router has started the network, the initiator joins the network (Router). The initiator then collects endpoint and cluster information from the Lighting device(s) on the Router node, and stores this information in a local lighting database.

Once the network (consisting of the initiator and one Router) is up and running, further nodes may be added as described in Section 44.4.2.



Creating a Network |

Parent topic: Using Touchlink

Adding to an existing network A network (that has been set up as described in Section 44.4.1) can be extended by adding a node. The Touchlink extension process is described below and illustrated in the figure below (also refer to the command list in Table 1).

Note: Received Touchlink requests and responses are handled as ZigBee PRO events. The event handling is not detailed below but is outlined in Section 44.6.

1. **Scan Request:** The initiator sends a Scan Request to nodes in its vicinity. The required function is:

- **eCLD_ZllCommissionCommandScanReqCommandSend()**

2. **Scan Response:** A receiving node replies to the Scan Request by sending a Scan Response. The required function is:

- **eCLD_ZllCommissionCommandScanRspCommandSend()**

3. **Device Information Request:** The initiator sends a Device Information Request to those detected nodes that are of interest. The required function is:

- **eCLD_ZllCommissionCommandDeviceInfoReqCommandSend()**

4. **Device Information Response:** A receiving node replies to the Device Information Request by sending a Device Information Response. The required function is:

- **eCLD_ZllCommissionCommandDeviceInfoRspCommandSend()**

5. **Identify Request (Optional):** The initiator may send an Identify Request to the node which has been chosen to be added to the network, in order to confirm that the correct physical node is being commissioned. The required function is:

- **eCLD_ZllCommissionCommandDeviceIdentifyReqCommandSend()**

6. **Network Join Request:** Depending on the target node type, the initiator sends a Network Join Router Request or Network Join End Device Request, as appropriate, to the target node. The required function is one of:

- **eCLD_ZllCommissionCommandNetworkJoinRouterReqCommandSend()**
- **eCLD_ZllCommissionCommandNetworkJoinEndDeviceReqCommandSend()**

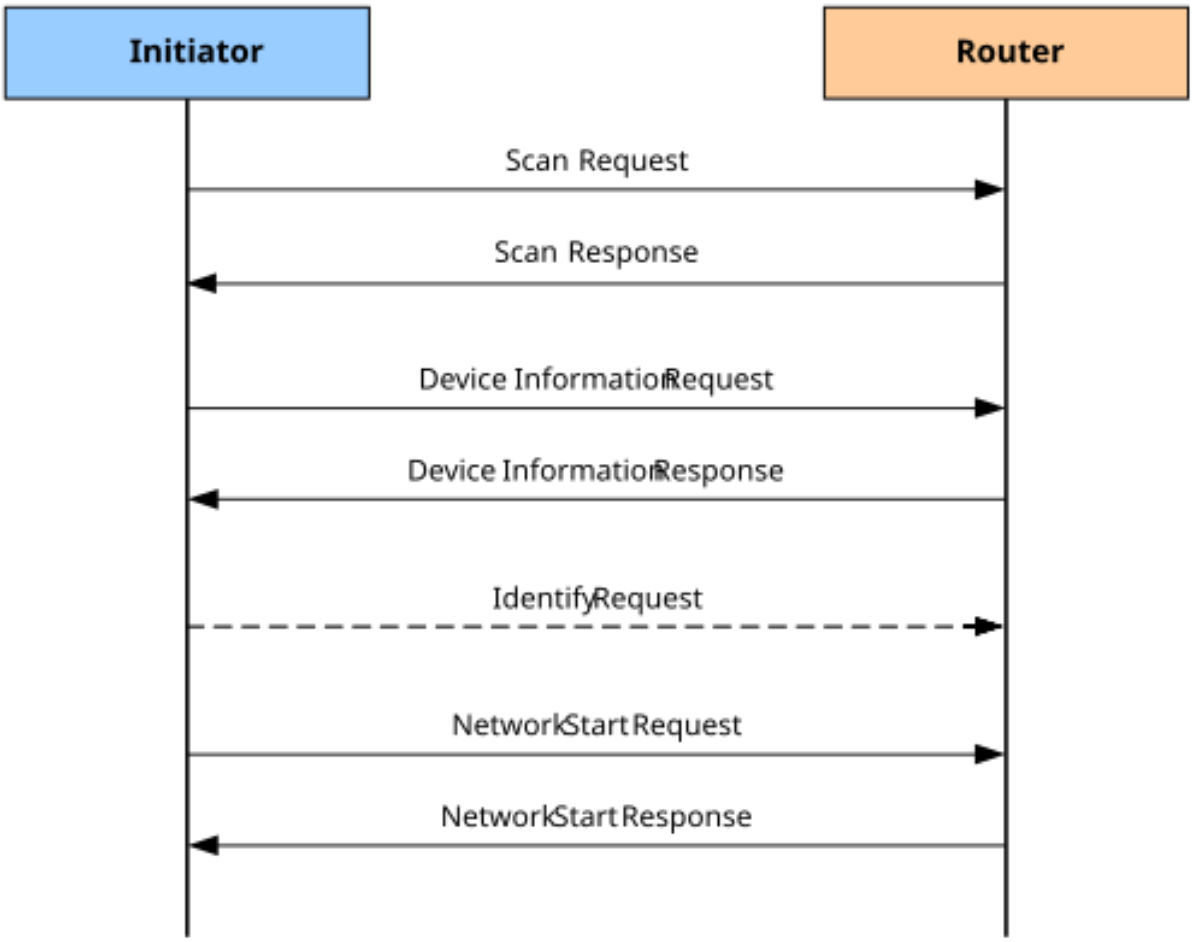
7. **Network Join Response:** Depending on the receiving node type, the node replies to the join request by sending a Network Join Router Response or Network Join End Device Response. The required function is one of:

- **eCLD_ZllCommissionCommandNetworkJoinRouterRspCommandSend()**
- **eCLD_ZllCommissionCommandNetworkJoinEndDeviceRspCommandSend()**

The node should now be a member of the network. The initiator then collects endpoint and cluster information from any Lighting device(s) on the new node, and stores this information in its local lighting database.

If the new node is to be used to control the light nodes of the network then it will need to learn certain information (such as controlled endpoints and configured groups) from the initiator - this is done using the Commissioning Utility commands, as described in Section 44.5.

Extending a Network (Adding a Node)



Parent topic: Using Touchlink

Updating network settings If one or more of the network settings change (e.g. the radio channel used), all nodes of the network need to be updated with the new settings.

To allow nodes to keep track of the status of the network settings, the Network Update Identifier is used. This identifier takes a value in the range 0x00 to 0xFF and is incremented when a network update has occurred (the value wraps around at 0xFF).

A node can be instructed to update its network settings by sending a Network Update Request to it. The required function is:

eCLD_ZllCommissionCommandNetworkUpdateReqCommandSend()

The payload of the sent command contains the latest network settings and the current value of the Network Update Identifier (see Section 44.8.17). If the payload value is more recent than the value held by the target node, the node should update its network settings with those in the payload.

Parent topic: Using Touchlink

Stealing a node A node that is already part of a network can be taken or 'stolen' by another network using Touchlink (in which case, the stolen node will cease to be a member of its previous network). This transfer can only be performed on a node which supports one or more Lighting devices (and not Controller devices).

The node is stolen using an initiator in the new network, e.g. from a remote control unit. The 'stealing' process is as follows:

1. The initiator sends a Scan Request to nodes in its vicinity. The required function is:

- **eCLD_ZllCommissionCommandScanReqCommandSend()**

2. A receiving node replies to the Scan Request by sending a Scan Response. The required function is:

- **eCLD_ZllCommissionCommandScanRspCommandSend()**

3. The initiator receives Scan Responses from one or more nodes and, based on these responses, selects a node (containing a Lighting device) that is already a member of another network.
4. The initiator then sends a Reset To Factory New Request to the desired node. The required function is:

- **eCLD_ZllCommissionCommandFactoryResetReqCommandSend()**

5. On receiving this request on the target node, the event E_CLD_COMMISSION_CMD_FACTORY_RESET_REQ is generated and the function **ZPS_eAplZdoLeaveNetwork()** should be called. In addition, all persistent data should be reset.
6. The node can then be commissioned into the new network by following the process in Section 44.4.2 from Step3.

Alternatively, instead of following the above process, a node can be stolen by either:

- Following the full process for creating a network in Section 44.4.1 and calling **ZPS_eAplZdoLeaveNetwork()**on the target node when a Network Start Request is received.
- Following the full process for adding a node in Section 44.4.2 and calling **ZPS_eAplZdoLeaveNetwork()**on the target node when a Network Join Router Request or Network Join End Device Request is received.

Note: If a node containing a Controller device (e.g. a remote control unit) is to be used in another network, it must first be reset using a Reset To Factory New Request. It can then be used to create a new network (see Section 44.4.1) or to learn the control information of an existing network (see Section 44.5).

Parent topic: Using Touchlink

Parent topic:*Touchlink Commissioning Cluster*

Using the Commissioning Utility The Commissioning Utility is used when a network node needs to learn lighting control information (such as controlled endpoints and configured groups) from another node in the network. It is typically used when a new remote control unit is introduced into the network and needs to learn information from an existing remote control unit.

Unlike Touchlink, the Commissioning Utility can be incorporated in the main application on the node (and therefore use the same endpoint). This requires:

- a Touchlink Commissioning cluster instance as a client to be created on the endpoint on the 'learner' node
- a Touchlink Commissioning cluster instance as a server to be created on the endpoint on the 'teacher' node

A Touchlink Commissioning cluster instance for the Commissioning Utility can be created using the function **eCLD_ZllUtilityCreateUtility()**, on both nodes.

It is the responsibility of the learner node to request the required information from the teacher node. The Commissioning Utility command set is summarised in *Table 85*. Commissioning Utility functions for issuing these commands are provided and are detailed in Section 44.7.2.

Command	Identifier	Description
Endpoint information	0x40	Sends information about local endpoint (from teacher to learner)
Get Group Identifiers Request	0x41	Requests Group information from a remote node (from learner to teacher)
Get Endpoint List Request	0x42	Requests endpoint information from a remote node (from learner to teacher)

Use of the above commands and associated functions is described below and is illustrated in the figure below.

Note: Received Commissioning Utility requests and responses are handled as ZigBee PRO events by the ZCL (this event handling is therefore transparent to the application).

1. **Endpoint Information command:** The teacher node first sends an Endpoint Information command containing basic information about its local endpoint (IEEE address, network address endpoint number, Profile ID, Device ID) to the learner node. The required function is:

- **eCLD_ZllUtilityCommandEndpointInformationCommandSend()**
- Note that the teacher node will already have the relevant target endpoint on the learner node from the joining process (described in Section 44.4).

2. **Get Endpoint List Request:** The learner node then sends a Get Endpoint List Request to the teacher node to request information about the remote endpoints that the teacher node controls. The required function is:

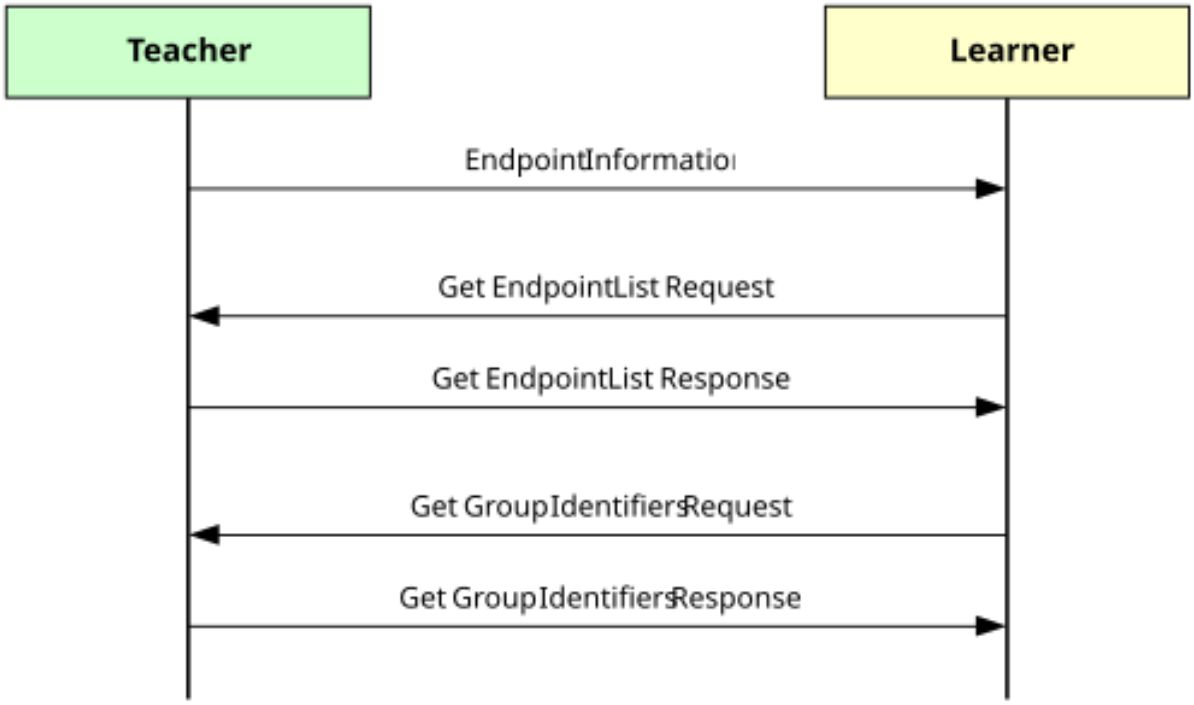
- **eCLD_ZllUtilityCommandGetEndpointListReqCommandSend()**
- The teacher node automatically replies to the Get Endpoint List Request by sending a Get Endpoint List Response containing the requested information.

3. **Get Group Identifiers Request:** The learner node then sends a Get Group Identifiers Request to the teacher node to request a list of the lighting groups configured on the teacher node. The required function is:

- **eCLD_ZllUtilityCommandGetGroupIdReqCommandSend()**

- The teacher node automatically replies to the Get Group Identifiers Request by sending a Get Group Identifiers Response containing the requested information.

Learning Process



To complete the learning process, the learner node may need other information which can be acquired using commands/functions of the relevant cluster.

Parent topic:*Touchlink Commissioning Cluster*

Touchlink Commissioning events Touchlink Commissioning cluster events that result from receiving Touchlink requests and responses must be handled at the application level (while events that result from Commissioning Utility requests and responses are handled by the ZCL).

When a Touchlink request or response command (e.g. a Scan Request) is received by a node, a stack event is generated which is wrapped in a tsZCL_CallBackEvent structure. In this structure:

- eEventType field is set to E_ZCL_CBET_CLUSTER_CUSTOM
- sClusterCustomMessage field's tsZCL_ClusterCustomMessage structure is filled in by:
  - setting u16ClusterId to ZLL_CLUSTER_ID_COMMISSIONING
  - pointing pvCustomData to the payload data of the received command
    - The above structure is described in Section 6.1.15.

The payload data contains a command ID, which uses one of the enumerations listed in Section 44.6.1. The event is passed to the ZCL event handler which checks that the command ID is valid for the target endpoint. If it is valid, the user-defined callback function is invoked that was specified through the function **eZLL_RegisterCommissionEndPoint()**. The callback function can access the payload through the tsCLD_ZllCommissionCustomDataStructure structure, which is created when the above function is called.

Thus, the above user-defined callback function must be designed to handle the relevant Touchlink events:

- For a request, the callback function may need to populate a structure with the required data and send a response using the appropriate response function, e.g. by calling **eCLD_ZllCommissionCommandScanRspCommandSend()** to respond to a Scan Request.

- For a response, the callback function may just need to extract the returned data from the event.

Alternatively, the callback function may simply notify the main application of the received command and provide the payload, so that the application can process the command.

Touchlink command events The events that can be generated for Touchlink are listed and described below (the enumerations are defined in the structure teCLD_ZllCommission_Command, shown in Section 44.9.1).

Event	Description
E_CLD_COMMISSION_CMD_SCAN_REQ	A Scan Request has been received (by server)
E_CLD_COMMISSION_CMD_SCAN_RSP	A Scan Response has been received (by client)
E_CLD_COMMISSION_CMD_DEVICE_INFO_	A Device Information Request has been received (by server)
E_CLD_COMMISSION_CMD_DEVICE_INFO_	A Device Information Response has been received (by client)
E_CLD_COMMISSION_CMD_IDENTIFY_REC	An Identify Request has been received (by server)
E_CLD_COMMISSION_CMD_FACTORY_RES	A Reset To Factory New Request has been received (by server)
E_CLD_COMMISSION_CMD_NETWORK_ST	A Network Start Request has been received (by server)
E_CLD_COMMISSION_CMD_NETWORK_ST	A Network Start Response has been received (by cli-ent)
E_CLD_COMMISSION_CMD_NETWORK_JO	

ROUTER_REQ

|A Network Join Router Request has been received (by server)| |E_CLD_COMMISSION_CMD_NETWORK_JOIN_

ROUTER_RSP

|A Network Join Router Response has been received (by client)| |E_CLD_COMMISSION_CMD_NETWORK_JOIN_

END_DEVICE_REQ

|A Network Join End Device Request has been received (by server)| |E_CLD_COMMISSION_CMD_NETWORK_JOIN_

END_DEVICE_RSP

|A Network Join End Device Response has been received (by client)| |E_CLD_COMMISSION_CMD_NETWORK_UPDATE_REQ|A Network Update Request has been received (by server)|

Parent topic: Touchlink Commissioning events

Commissioning Utility Command Events The events that can be generated for the Commissioning Utility are listed and described below (the enumerations are defined in the structure teCLD_ZllUtility_Command, shown in Section 44.9.2).

Event	Description
E_CLD_UTILITY_CMD_ENDP	An Endpoint Information command has been received (by client)
E_CLD_UTILITY_CMD_GET_(	A Get Group Identifiers Request has been received (by server) or a Get Group Identifiers Response has been received (by client)
E_CLD_UTILITY_CMD_GET_I	A Get Endpoint List Request has been received (by server) or a Get Endpoint List Response has been received (by client)

Parent topic: Touchlink Commissioning events

Parent topic:*Touchlink Commissioning Cluster*

Functions The functions of the Touchlink Commissioning cluster are divided into two categories:

- Touchlink functions, detailed in Section 44.7.1
- Commissioning Utility functions, detailed in Section 44.7.2

Touchlink functions The following Touchlink functions are provided:

1. eZLL_RegisterCommissionEndPoint
2. eCLD_ZllCommissionCreateCommission
3. eCLD_ZllCommissionCommandScanReqCommandSend
4. eCLD_ZllCommissionCommandScanRspCommandSend
5. eCLD_ZllCommissionCommandDeviceInfoReqCommandSend
6. eCLD_ZllCommissionCommandDeviceInfoRspCommandSend
7. eCLD_ZllCommissionCommandDeviceIdentifyReqCommandSend
8. eCLD_ZllCommissionCommandFactoryResetReqCommandSend
9. eCLD_ZllCommissionCommandNetworkStartReqCommandSend
10. eCLD_ZllCommissionCommandNetworkStartRspCommandSend
11. eCLD_ZllCommissionCommandNetworkJoinRouterReqCommandSend
12. eCLD_ZllCommissionCommandNetworkJoinRouterRspCommandSend
13. eCLD_ZllCommissionCommandNetworkJoinEndDeviceReqCommandSend
14. eCLD_ZllCommissionCommandNetworkJoinEndDeviceRspCommandSend
15. eCLD_ZllCommissionCommandNetworkUpdateReqCommandSend

eZLL_RegisterCommissionEndPoint

```
teZCL_Status eZLL_RegisterCommissionEndPoint(
    uint8 u8EndPointIdentifier,
    tfpZCL_ZCLCallBackFunction cbCallBack,
    tsZLL_CommissionEndpoint *psDeviceInfo);
```

Description This function registers a 'commissioning' endpoint for Touchlink and creates a Touchlink Commissioning cluster instance on the endpoint.

Touchlink must have its own application (separate from the main application) on its own endpoint.

This function uses **eCLD_ZllCommissionCreateCommission()** to create the cluster instance. The type of cluster instance to be created (server or client, or both) is determined using the compile-time options in the header file **zcl_options.h** (refer to Section 44.10).

Parameters

- *u8EndPointIdentifier* Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
- *cbCallBack* Pointer to a callback function to handle events associated with the registered endpoint
- *psDeviceInfo* Pointer to structure to be used to hold Touchlink endpoint information (see Section 44.8.1)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCreateCommission

```
teZCL_Status eCLD_ZllCommissionCreateCommission(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvSharedStructPtr,
    tsZCL_AttributeStatus *psAttributeStatus,
    tsCLD_ZllCommissionCustomDataStructure
*psCustomDataStructure);
```

Description This function creates a Touchlink Commissioning cluster instance for Touchlink on the endpoint of the calling application. The type of cluster instance (server or client) to be created must be specified.

In practice, this function does not need to be called explicitly by the application, as the function **eZLL_RegisterCommissionEndPoint()** calls this function to create the cluster instance.

Parameters

- *psClusterInstance* Pointer to cluster instance structure on local endpoint
- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition* Pointer to cluster definition structure containing information about the cluster
- *pvSharedStructPtr* Pointer to structure containing the shared storage for the cluster
- *psAttributeStatus* Pointer to a structure containing the storage for each attribute's status

- *psCustomDataStructure* Pointer to custom data to be provided to the cluster (see Section 44.8.3)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCommandScanReqCommandSend

```
eCLD_ZllCommissionCommandScanReqCommandSend(
    ZPS_tsInterPanAddress *psDestinationAddress,
     *pu8TransactionSequenceNumber,
    tsCLD_ZllCommission_ScanReqCommandPayload
    *psPayload);
```

Description This function is used to send a Scan Request command to initiate a scan for other nodes in the local neighbourhood. The command is sent as an inter-PAN message.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *psDestinationAddress* Pointer to stucture containing PAN ID and address information for target node(s)
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to structure containing payload data for the Scan Request command (see Section44.8.5)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCommandScanRspCommandSend

```
PUBLIC teZCL_Status eCLD_ZllCommissionCommandScanRspCommandSend(
    ZPS_tsInterPanAddress *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ZllCommission_ScanRspCommandPayload
*psPayload);
```

Description This function is used to send a Scan Response command containing information about the local node in reply to a received Scan Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

- *psDestinationAddress* Pointer to stucture containing PAN ID and address information for target node
- *pu8TransactionSequenceNumber* Pointer to location containing the Transaction Sequence Number (TSN) of the response
- *psPayload* Pointer to structure containing payload data for the Scan Response command (see Section 44.8.6)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCommandDeviceInfoReqCommandSend

```
teZCL_Status eCLD_ZllCommissionCommandDeviceInfoReqCommandSend(
ZPS_tsInterPanAddress *psDestinationAddress,
uint8 *pu8TransactionSequenceNumber, tsCLD_ZllCommission_DeviceInfoReqCommandPayload
↪*psPayload);
```

Description This function is used to send a Device Information Request command to obtain information about the devices on a remote node. The command is sent as an inter-PAN message.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *psDestinationAddress* Pointer to stucture containing PAN ID and address information for target node
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to structure containing payload data for the Device Information Request command (see Section 44.8.7)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCommandDeviceInfoRspCommandSend

```
PUBLIC teZCL_Status eCLD_ZllCommissionCommandDeviceInfoRspCommandSend(
ZPS_tsInterPanAddress *psDestinationAddress,
uint8 *pu8TransactionSequenceNumber,
tsCLD_ZllCommission_DeviceInfoRspCommandPayload
*psPayload);
```

Description This function is used to send a Device Information Response command containing information about the devices on the local node in reply to a received Device Information Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

- *psDestinationAddress* Pointer to stucture containing PAN ID and address information for target node
- *pu8TransactionSequenceNumber* Pointer to location containing the Transaction Sequence Number (TSN) of the response
- *psPayload* Pointer to structure containing payload data for the Device Information Response command (see Section 44.8.8)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCommandDeviceIdentifyReqCommandSend

```
teZCL_Status eCLD_ZllCommissionCommandDeviceIdentifyReqCommandSend(
ZPS_tsInterPanAddress *psDestinationAddress,
uint8 *pu8TransactionSequenceNumber,
tsCLD_ZllCommission_IdentifyReqCommandPayload
*psPayload);
```

Description This function is used to send an Identify Request command to ask a remote node to identify itself by entering 'identify mode' (this is a visual indication, such as flashing a LED). The command is sent as an inter-PAN message.

The command payload contains a value indicating the length of time, in seconds, that the target device should remain in identify mode. It is also possible to use this command to instruct the target node to immediately exit identify mode (if it is already in this mode).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *psDestinationAddress* Pointer to stucture containing PAN ID and address information for target node
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to structure containing payload data for the Identify Request command (see Section 44.8.9)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCommandFactoryResetReqCommandSend

```
teZCL_Status eCLD_ZllCommissionCommandFactoryResetReqCommandSend(
    ZPS_tsInterPanAddress *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ZllCommission_FactoryResetReqCommandPayload
    *psPayload);
```

Description This function is used to send a Reset to Factory New Request command to ask a remote node to return to its 'factory new' state. The command is sent as an inter-PAN message.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *psDestinationAddress* Pointer to stucture containing PAN ID and address information for target node
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to structure containing payload data for the Reset to Factory New Request command (see Section 44.8.10)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCommandNetworkStartReqCommandSend

```
teZCL_Status eCLD_ZllCommissionCommandNetworkStartReqCommandSend(
    ZPS_tsInterPanAddress *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ZllCommission_NetworkStartReqCommandPayload
*psPayload);
```

Description This function is used to send a Network Start Request command to create a new network with a detected Router. The command is sent as an inter-PAN message.

The function is called once the results of a Scan Request command have been received and a detected Router has been selected.

The command payload contains information about the network and the local node, as well as certain data for the target node. This payload information is detailed in Section 44.8.11.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *psDestinationAddress* Pointer to stucture containing PAN ID and address information for target node
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to structure containing payload data for the Network Start Request command (see Section 44.8.11)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCommandNetworkStartRspCommandSend

```
PUBLIC teZCL_Status eCLD_ZllCommissionCommandNetworkStartRspCommandSend(
    ZPS_tsInterPanAddress *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ZllCommission_NetworkStartRspCommandPayload
    *psPayload);
```

Description This function is used to send a Network Start Response command to confirm that the local (Router) node is ready to be the first node to join a newly created network in reply to a received Network Start Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

- *psDestinationAddress* Pointer to stucture containing PAN ID and address information for target node
- *pu8TransactionSequenceNumber* Pointer to location containing the Transaction Sequence Number (TSN) of the response
- *psPayload* Pointer to structure containing payload data for the Network Start Response command (see Section 44.8.12)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCommandNetworkJoinRouterReqCommandSend

```
teZCL_Status eCLD_ZllCommissionCommandNetworkJoinRouterReqCommandSend(
    ZPS_tsInterPanAddress *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ZllCommission_NetworkJoinRouterReqCommandPayload
    *psPayload);
```

Description This function is used to send a Network Join Router Request command to allow a detected Router to join the created network. The command is sent as an inter-PAN message.

The function can be called once a network has been created. The target Router is distinct from the Router that was included when network was created.

The command payload contains information about the network and the local node, as well as certain data for the target node. This payload information is detailed in Section 44.8.13.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *psDestinationAddress* Pointer to stucture containing PAN ID and address information for target node
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to structure containing payload data for the Network Join Router Request command (see Section 44.8.13)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCommandNetworkJoinRouterRspCommandSend

```
PUBLIC teZCL_Status eCLD_ZllCommissionCommandNetworkJoinRouterRspCommandSend(
    ZPS_tsInterPanAddress psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ZllCommission_NetworkJoinRouterRspCommandPayload
    *psPayload);
```

Description This function is used to send a Network Join Router Response command to confirm that the local (Router) node is ready to join a network in reply to a received Network Join Router Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

- *psDestinationAddress* Pointer to stucture containing PAN ID and address information for target node
- *psDestinationAddress* Pointer to location containing the Transaction Sequence Number (TSN) of the response
- **psPayload* Pointer to structure containing payload data for the Network Join Router Response command (see Section 44.8.14)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCommandNetworkJoinEndDeviceReqCommandSend

```
teZCL_Status    eCLD_ZllCommissionCommandNetworkJoinEndDeviceReqCommandSend(
    ZPS_tsInterPanAddress *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ZllCommission_NetworkJoinEndDeviceReqCommandPayload
*psPayload);
```

Description This function is used to send a Network Join End Device Request command to allow a detected End Device to join the created network. The command is sent as an inter-PAN message.

The function can be called once a network has been created.

The command payload contains information about the network and the local node, as well as certain data for the target node. This data includes a range of network addresses and a range of group IDs from which the target End Device can assign values to the other nodes - in this case, the End Device would typically be a remote control unit. This payload information is detailed in Section 44.8.15.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *psDestinationAddress*: Pointer to stucture containing PAN ID and address information for target node
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

- *psPayload*: Pointer to structure containing payload data for the Network Join End Device Request command (see Section 44.8.15)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCommandNetworkJoinEndDeviceRspCommandSend

```
PUBLIC teZCL_Status eCLD_ZllCommissionCommandNetworkJoinEndDeviceRspCommandSend(
ZPS_tsInterPanAddress *psDestinationAddress,
uint8 *pu8TransactionSequenceNumber,
tsCLD_ZllCommission_NetworkJoinEndDeviceRspCommandPayload
*psPayload);
```

Description This function is used to send a Network Join End Device Response command to confirm that the local (End Device) node is ready to join a network in reply to a received Network Join End Device Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

- *psDestinationAddress* Pointer to stucture containing PAN ID and address information for target node
- *pu8TransactionSequenceNumber* Pointer to location containing the Transaction Sequence Number (TSN) of the response
- *psPayload* Pointer to structure containing payload data for the Network Join End Device Response command (see Section 44.8.16)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

eCLD_ZllCommissionCommandNetworkUpdateReqCommandSend

```
teZCL_Status eCLD_ZllCommissionCommandNetworkUpdateReqCommandSend(
        ZPS_tsInterPanAddress *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllCommission_NetworkUpdateReqCommandPayload
        *psPayload);
```

Description This function is used to send a Network Update Request command to bring a node that has missed a network update back into the network. The command is sent as an inter-PAN message.

The command payload contains information about the network, including the current value of the Network Update Identifier. This identifier takes a value in the range 0x00 to 0xFF and is incremented when a network update has occurred (the value wraps around at 0xFF). Thus, if this value in the payload is more recent than the value of this identifier held by the target node, the node should update its network settings using the values in the rest of the payload. The payload information is detailed in Section 44.8.17.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *psDestinationAddress* Pointer to stucture containing PAN ID and address information for target node
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to structure containing payload data for the Network Update Request command (see Section 44.8.17)

Returns

- E_ZCL_SUCCESS

Parent topic: Touchlink functions

Parent topic: Functions

Commissioning Utility functions The following Commissioning Utility functions are provided:

1. eCLD_ZllUtilityCreateUtility
2. eCLD_ZllUtilityCommandEndpointInformationCommandSend
3. eCLD_ZllUtilityCommandGetGroupIdReqCommandSend
4. eCLD_ZllUtilityCommandGetGroupIdRspCommandSend
5. eCLD_ZllUtilityCommandGetEndpointListReqCommandSend
6. eCLD_ZllUtilityCommandGetEndpointListRspCommandSend
7. eCLD_ZllUtilityCommandHandler

eCLD_ZllUtilityCreateUtility

```
teZCL_Status eCLD_ZllUtilityCreateUtility(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvSharedStructPtr,
        tsZCL_AttributeStatus psAttributeStatus,
        tsCLD_ZllUtilityCustomDataStructure
    *psCustomDataStructure);
```

Description This function creates a Touchlink Commissioning cluster instance for the Commissioning Utility. The cluster instance is created on the endpoint of the calling application, which should be the main application on the node. The type of cluster instance (server or client) to be created must be specified.

Parameters

- *psClusterInstance* Pointer to cluster instance structure on local endpoint
- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition* Pointer to cluster definition structure containing information about the cluster
- *pvSharedStructPtr* Pointer to structure containing the shared storage for the cluster
- *psAttributeStatus* Pointer to a structure containing the storage for each attribute's status
- *psCustomDataStructure* Pointer to custom data to be provided to the cluster (see Section 44.8.20)

Returns

- E_ZCL_SUCCESS

Parent topic: Commissioning Utility functions

eCLD_ZllUtilityCommandEndpointInformationCommandSend

```
teZCL_Status eCLD_ZllUtilityCommandEndpointInformationCommandSend(
        uint8 u8SrcEndpoint,
        uint8 u8DstEndpoint,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllUtility_EndpointInformationCommandPayload
*psPayload);
```

Description This function is used to send an Endpoint Information command to provide a remote endpoint with general information about the local endpoint (this may prompt the remote endpoint to request further information about the local endpoint). The function would typically be used to send local endpoint information from a 'teacher' node to a 'learner' node, in order to facilitate two-way communication between the Commissioning Utilities on the two nodes.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the command. The TSN in the response is set to match the specified TSN, allowing an incoming response to be paired with the original command. This is useful when sending more than one command to the same destination endpoint.

Parameters

- *u8SrcEndpoint* Number of local endpoint (1-240)
- *u8DstEndpoint* Number of destination endpoint (1-240)
- *psDestinationAddress* Pointer to stucture containing address information for target node

- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the command
- *psPayload* Pointer to structure to contain payload data for the Endpoint Information command (see Section 44.8.20)

Returns

- E_ZCL_SUCCESS

Parent topic: Commissioning Utility functions

eCLD_ZllUtilityCommandGetGroupIdReqCommandSend

```
teZCL_Status eCLD_ZllUtilityCommandGetGroupIdReqCommandSend(
        uint8 u8Srcendpoint,
        uint8 u8DstEndpoint,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8StartIndex);
```

Description This function is used to send a Get Group Identifiers Request command to obtain information about the groups (of lights) that have been configured on a remote endpoint. The function would typically be used on a 'learner' node to request the groups that have been configured on a 'teacher' node.

The first group from the groups list to be included in the returned information must be specified in terms of an index.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SrcEndpoint* Number of local endpoint (1-240)
- *u8DstEndpoint* Number of destination endpoint (1-240)
- *psDestinationAddress* Pointer to stucture containing address information for target node
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8StartIndex* Index in group list of the first group to include in the returned information

Returns

- E_ZCL_SUCCESS

Parent topic: Commissioning Utility functions

eCLD_ZllUtilityCommandGetGroupIdRspCommandSend

```
PUBLIC teZCL_Status   eCLD_ZllUtilityCommandGetGroupIdRspCommandSend(
        uint8 u8SrcEndpoint,
        uint8 u8DstEndpoint,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8StartIndex);
```

Description This function is used to send a Get Group Identifiers Response command containing information about the groups (of lights) that have been configured on the local endpoint. The function would typically be used on a 'teacher' node to respond to a Get Group Identifiers Request from a 'learner' node.

The first group from the groups list to be included in the returned information must be specified in terms of an index. The returned information includes this index, the number of (consecutive) groups included and the identifier of each group.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

- *u8SrcEndpoint* Number of local endpoint (1-240)
- *u8DstEndpoint* Number of destination endpoint (1-240)
- *psDestinationAddress* Pointer to stucture containing address information for target node
- *pu8TransactionSequenceNumber* Pointer to location containing the Transaction Sequence Number (TSN) of the response
- *u8StartIndex* Index in group list of the first group to include in the returned information

Returns

- E_ZCL_SUCCESS

Parent topic: Commissioning Utility functions

eCLD_ZllUtilityCommandGetEndpointListReqCommandSend

```
teZCL_Status eCLD_ZllUtilityCommandGetEndpointListReqCommandSend(
        uint8 u8SrcEndpoint,
        uint8 u8DstEndpoint,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8StartIndex);
```

Description This function is used to send a Get Endpoint List Request command to obtain information about controlled endpoints. The function would typically be used on a 'learner' node to request the remote endpoints that a 'teacher' node controls.

The first endpoint from the endpoints list to be included in the returned information must be specified in terms of an index.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing

an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SrcEndpoint* Number of local endpoint (1-240)
- *u8DstEndpoint* Number of destination endpoint (1-240)
- *psDestinationAddress* Pointer to stucture containing address information for target node
- *pu8TransactionSequenceNumber* Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8StartIndex* Index in endpoint list of the first endpoint to include in the returned information

Returns

- E_ZCL_SUCCESS

Parent topic: Commissioning Utility functions

eCLD_ZllUtilityCommandGetEndpointListRspCommandSend

```
PUBLIC teZCL_Status eCLD_ZllUtilityCommandGetEndpointListRspCommandSend(
        uint8 u8SrcEndpoint,
        uint8 u8DstEndpoint,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8StartIndex);
```

Description This function is used to send a Get Endpoint List Response command containing information about controlled endpoints. The function would typically be used on a 'teacher' node to respond to a Get Endpoint List Request from a 'learner' node.

The first endpoint from the endpoints list to be included in the returned information must be specified in terms of an index. The returned information will include this index, the number of (consecutive) endpoints included and the information about each endpoint (including endpoint number, identifier of resident ZigBee device and version of this device).

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

- *u8SrcEndpoint* Number of local endpoint (1-240)
- *u8DstEndpoint* Number of destination endpoint (1-240)
- *psDestinationAddress* Pointer to stucture containing address information for target node
- *pu8TransactionSequenceNumber* Pointer to location containing the Transaction Sequence Number (TSN) of the response
- *u8StartIndex* Index in endpoint list of the first endpoint to include in the returned information

Returns

- E_ZCL_SUCCESS

Parent topic: Commissioning Utility functions

eCLD_ZllUtilityCommandHandler

```
teZCL_Status eCLD_ZllUtilityCommandHandler(
    ZPS_tsAfEvent *pZPSevent,
    tsZCL_EndPointDefinition *psEndPointDefinition,
    tsZCL_ClusterInstance *psClusterInstance);
```

Description This function parses a ZigBee PRO event and invokes the user-defined callback function that has been registered for the device (using the relevant endpoint registration function).

The registered user-defined callback function must be designed to handle events associated with the Commissioning Utility.

Parameters

- *pZPSevent* Pointer to received ZigBee PRO event
- *psEndPointDefinition* Pointer to structure which defines endpoint on which the Commissioning Utility resides
- *psClusterInstance* Pointer to Touchlink Commissioning cluster instance structure

Returns

- E_ZCL_SUCCESS

Parent topic: Commissioning Utility functions

Parent topic: Functions

Parent topic:*Touchlink Commissioning Cluster*

Structures This section details the structures used in the Touchlink Commissioning cluster (both Touchlink and Commissioning Utility parts).

tsZLL_CommissionEndpoint This structure is used to hold endpoint information for a Touchlink application.

```
typedef struct
{
tsZCL_EndPointDefinition sEndPoint;
tsZLL_CommissionEndpointClusterInstances sClusterInstance;
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
    tsCLD_ZllCommission sZllCommissionServerCluster;
    tsCLD_ZllCommissionCustomDataStructure
                sZllCommissionServerCustomDataStructure;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
    tsCLD_ZllCommission sZllCommissionClientCluster;
    tsCLD_ZllCommissionCustomDataStructure
                sZllCommissionClientCustomDataStructure;
```

(continues on next page)

(continued from previous page)

```
#endif
} tsZLL_CommissionEndpoint;
```

where:

- sEndPoint is a ZCL structure containing information about the endpoint (refer to Section 6.1.1).
- sClusterInstance is a structure containing information about the Touchlink Commissioning cluster instance on the endpoint (see Section 44.8.2).
- For a Touchlink server, the following fields are used:
  - sZllCommissionServerCluster is the Touchlink Commissioning cluster structure (which contains no attributes).
  - sZllCommissionServerCustomDataStructure is a structure containing custom data for the cluster server (see Section 44.8.3).
- For a Touchlink client, the following fields are used:
  - sZllCommissionClientClusteris the Touchlink Commissioning cluster structure (which contains no attributes).
  - sZllCommissionClientCustomDataStructure is a structure containing custom data for the cluster client (see Section 44.8.3).

Parent topic: Structures

tsZLL_CommissionEndpointClusterInstances This structure holds information about the Touchlink Commissioning cluster instance on an endpoint.

```
typedef struct PACK
{
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
    tsZCL_ClusterInstance sZllCommissionServer;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
    tsZCL_ClusterInstance sZllCommissionClient;
#endif
} tsZLL_CommissionEndpointClusterInstances;
```

where:

- s``ZllCommissionServer is a ZCL structure containing information about the Touchlink Commissioning cluster server instance (refer to Section 6.1.16).
- s``ZllCommissionClient is a ZCL structure containing information about the Touchlink Commissioning cluster client instance (refer to Section 6.1.16).

Parent topic: Structures

tsCLD_ZllCommissionCustomDataStructure This structure is used to hold the data for a Touchlink command received by a node.

```
typedef struct
{
    tsZCL_ReceiveEventAddressInterPan     sRxInterPanAddr;
    tsZCL_CallBackEvent                   sCustomCallBackEvent;
    tsCLD_ZllCommissionCallBackMessage    sCallBackMessage;
```

(continues on next page)

(continued from previous page)

```
} tsCLD_ZllCommissionCustomDataStructure;
```

where:

- RxInterPanAddr is a ZCL structure containing the Inter-PAN addresses of the source and destination nodes of the command.
- sCustomCallBackEvent is the ZCL event structure for the command.
- sCallBackMessage is a structure containing the command ID and payload (see Section 44.8.4).

Parent topic: Structures

tsCLD_ZllCommissionCallBackMessage This structure contains the command ID and payload for a received Touchlink command.

```
typedef struct
{
    uint8   u8CommandId;
    union
    {
        tsCLD_ZllCommission_ScanReqCommandPayload
                                *psScanReqPayload;
        tsCLD_ZllCommission_ScanRspCommandPayload
                                *psScanRspPayload;
        tsCLD_ZllCommission_IdentifyReqCommandPayload
                                *psIdentifyReqPayload;
        tsCLD_ZllCommission_DeviceInfoReqCommandPayload
                                *psDeviceInfoReqPayload;
        tsCLD_ZllCommission_DeviceInfoRspCommandPayload
                                *psDeviceInfoRspPayload;
        tsCLD_ZllCommission_FactoryResetReqCommandPayload
                                *psFactoryResetPayload;
        tsCLD_ZllCommission_NetworkStartReqCommandPayload
                                *psNwkStartReqPayload;
        tsCLD_ZllCommission_NetworkStartRspCommandPayload
                                *psNwkStartRspPayload;
        tsCLD_ZllCommission_NetworkJoinRouterReqCommandPayload
                                *psNwkJoinRouterReqPayload;
        tsCLD_ZllCommission_NetworkJoinRouterRspCommandPayload
                                *psNwkJoinRouterRspPayload;
        tsCLD_ZllCommission_NetworkJoinEndDeviceReqCommandPayload
                                *psNwkJoinEndDeviceReqPayload;
        tsCLD_ZllCommission_NetworkJoinEndDeviceRspCommandPayload
                                *psNwkJoinEndDeviceRspPayload;
        tsCLD_ZllCommission_NetworkUpdateReqCommandPayload
                                *psNwkUpdateReqPayload;
    } uMessage;
} tsCLD_ZllCommissionCallBackMessage;
```

where:

- u8CommandId is the command ID - enumerations are provided, as detailed in Section 44.6.1.
- uMessage contains the payload of the command, where the structure used depends on the command ID (the structures are detailed in the sections below).

Parent topic: Structures

tsCLD_ZllCommission_ScanReqCommandPayload This structure is used to hold the payload data for a Touchlink Scan Request command.

```
typedef struct
{
    uint32 u32TransactionId;
    uint8  u8ZigbeeInfo;
    uint8  u8ZllInfo;
} tsCLD_ZllCommission_ScanReqCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.
- u8ZigbeeInfo is a bitmap of ZigBee information which indicates the ZigBee device type of the sending node and whether the radio receiver remains on when the node is idle. This information is inserted by the ZigBee stack.
- u8ZllInfo is a bitmap indicating properties of the sending node, including whether the node is factory new, whether the node is able to assign addresses to other nodes and whether the node is able to initiate a link operation (supports Touchlink Commissioning cluster on the client side). This information is inserted automatically.

Parent topic: Structures

tsCLD_ZllCommission_ScanRspCommandPayload This structure is used to hold the payload data for a Touchlink Scan Response command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint8   u8RSSICorrection;
    uint8   u8ZigbeeInfo;
    uint8   u8ZllInfo;
    uint16  u16KeyMask;
    uint32  u32ResponseId;
    uint64  u64ExtPanId;
    uint8   u8NwkUpdateId;
    uint8   u8LogicalChannel;
    uint16  u16PanId;
    uint16  u16NwkAddr;
    uint8   u8NumberSubDevices;
    uint8   u8TotalGroupIds;
    uint8   u8Endpoint;
    uint16  u16ProfileId;
    uint16  u16DeviceId;
    uint8   u8Version;
    uint8   u8GroupIdCount;
} tsCLD_ZllCommission_ScanRspCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- u8RSSICorrection is the 8-bit RSSI correction offset for the node, in the range 0x00 to 0x20.
- u8ZigbeeInfo is an 8-bit field containing the following ZigBee-related information:
  - Bits 1-0: Node type (00 - Co-ordinator, 01 - Router, 10 - End Device)
  - Bit 2: Rx on when idle (1 - On, 0 - Off)

  - Bits 7-3: Reserved
- u8ZllInfo is an 8-bit field containing the following information:
  - Bit 0: Factory new (1 - Yes, 0 - No)
  - Bit 1: Address assignment capability (1 - Yes, 0 - No)
  - Bits 3-2: Reserved
  - Bit 4: Touchlink initiator (1 - Yes, 0 - No)
  - Bit 5: Touchlink priority request (1 - Yes, 0 - No)
  - Bits 7-6: Reserved
- u16KeyMask is a 16-bit bitmap indicating which link key is installed on the node - only one bit should be set to '1', corresponding to the key that is in use. The possible values and keys are:
  - 0x0001 (bit 0 set): Development key (defined by developer for use during application development)
  - 0x0010 (bit 4 set): Master key (obtained from the ZigBee Alliance after successful certification and agreement with the terms of the 'ZLL Security Key Licence and Confidentialty Agreement')
  - 0x8000 (bit 15 set): Certification key (defined in the ZLL Specification for use during development and during certification at test houses)
- u``32ResponseId is a 32-bit random identifier for the response, used during network key transfer.
- u64ExtPanId is the 64-bit Extended PAN ID of a network to which the node already belongs, if any (a zero value indicates no network membership).
- u8NwkUpdateId is the current value of the Network Update Identifier on the node (see Section 44.4.3).
- u8LogicalChannel is the number of the IEEE 802.15.4 radio channel used by a network to which the node already belongs, if any (a zero value indicates no network membership and therefore that no particular channel is used).
- u16PanId is the 16-bit PAN ID of a network to which the node already belongs, if any (a zero value indicates no network membership).
- u16NwkAddr is the 16-bit network address currently assigned to the node (the value 0xFFFF indicates that the node is 'factory new' and has no assigned network address).
- u8NumberSubDevices is the number of ZigBee devices on the node.
- u8TotalGroupIds is the total number of groups (of lights) supported on the node (across all devices).
- u8Endpoint is number of the endpoint (in the range 1-240) on which the ZigBee device is resident (this field is only used when there is only one ZigBee device on the node).
- u16ProfileId is the 16-bit identifier of the ZigBee application profile that is supported by the device (this field is only used when there is only one ZigBee device on the node).
- u16DeviceId is the 16-bit Device Identifier supported by the device (this field is only used when there is only one ZigBee device on the node).
- u8Version is an 8-bit version number for the device - the four least significant bits are from the Application Device Version field of the appropriate Simple Descriptor and the four most significant bits are zero (this field is only used when there is only one ZigBee device on the node).
- u8GroupIdCount is the number of groups (of lights) supported by the device (this field is only used when there is only one ZigBee device on the node).

Parent topic: Structures

tsCLD_ZllCommission_DeviceInfoReqCommandPayload This structure is used to hold the payload data for a Touchlink Device Information Request command.

```
typedef struct
{
    uint32 u32TransactionId;
    uint8  u8StartIndex;
} tsCLD_ZllCommission_DeviceInfoReqCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.
- u8StartIndex specifies the index (starting from 0) of the first entry in the device table from which device information should be obtained.

Parent topic: Structures

tsCLD_ZllCommission_DeviceInfoRspCommandPayload This structure is used to hold the payload data for a Touchlink Device Information Response command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint8   u8NumberSubDevices;
    uint8   u8StartIndex;
    uint8   u8DeviceInfoRecordCount;
    tsCLD_ZllDeviceRecord asDeviceRecords[ZLL_MAX_DEVICE_RECORDS];
} tsCLD_ZllCommission_DeviceInfoRspCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- u8NumberSubDevices is the number of ZigBee devices on the node (as reported in the Scan Response).
- u8StartIndex is the index (starting from 0) of the first entry in the device table from which device information has been obtained (this value should be as specified in the corresponding request).
- u8DeviceInfoRecordCount indicates the number of device information records included in the response (in the range 0 to 5).
- asDeviceRecords[] is an array, where each array element is a tsCLD_ZllDeviceRecord structure containing a device information record for one ZigBee device on the node.

Parent topic: Structures

tsCLD_ZllCommission_IdentifyReqCommandPayload This structure is used to hold the payload data for a Touchlink Identify Request command.

```
typedef struct
{
    uint32  u32TransactionId;
```

(continues on next page)

(continued from previous page)

```
    uint16  u16Duration;
} tsCLD_ZllCommission_IdentifyReqCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.
- u16Duration specifies the length of time (in seconds) that the target node is to remain in identify mode. The possible values are:
  - 0x0000: Exit identify mode immediately
  - 0x0001–0xFFFE: Number of seconds to remain in identify mode
  - 0xFFFF: Remain in identify mode for the default time for the target node
    - If the target node is unable to provide accurate timings, it will attempt to remain in identify mode for as close to the requested time as possible

Parent topic: Structures

tsCLD_ZllCommission_FactoryResetReqCommandPayload This structure is used to hold the payload data for a Touchlink Reset to Factory New Request command.

```
typedef struct
{
    uint32 u32TransactionId;
} tsCLD_ZllCommission_FactoryResetReqCommandPayload;
```

where u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.

Parent topic: Structures

tsCLD_ZllCommission_NetworkStartReqCommandPayload This structure is used to hold the payload data for a Touchlink Network Start Request command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint64  u64ExtPanId;
    uint8   u8KeyIndex;
    uint8   au8NwkKey[16];
    uint8   u8LogicalChannel;
    uint16  u16PanId;
    uint16  u16NwkAddr;
    uint16  u16GroupIdBegin;
    uint16  u16GroupIdEnd;
    uint16  u16FreeNwkAddrBegin;
    uint16  u16FreeNwkAddrEnd;
    uint16  u16FreeGroupIdBegin;
    uint16  u16FreeGroupIdEnd;
    uint64  u64InitiatorIEEEAddr;
    uint16  u16InitiatorNwkAddr;
} tsCLD_ZllCommission_NetworkStartReqCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.
- u64ExtPanId is the Extended PAN ID (EPID) of the new network (if set to zero, the target node will choose the EPID).
- u8KeyIndex is a value indicating the type of security key used to encrypt the randomly generated network key in au8NwkKey. The valid values are as follows (all other values are reserved for future use):
  - 0: Development key, used during development before ZigBee certification
  - 4: Master key, used after successful ZigBee certification
  - 15: Certification key, used during ZigBee certification testing
- au8NwkKey[16] is the 128-bit randomly generated network key encrypted using the key specified in u8KeyIndex.
- u8LogicalChannel is the number of the IEEE 802.15.4 radio channel to be used by the network (if set to zero, the target node will choose the channel).
- u16PanId is the PAN ID of the new network (if set to zero, the target node will choose the PAN ID).
- u16NwkAddr is the 16-bit network (short) address assigned to the target node
- u16GroupIdBegin is the start value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- u16GroupIdEnd is the end value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- u16FreeNwkAddrBegin is the start address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- u16FreeNwkAddrEnd is the end address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- u16FreeGroupIdBegin is the start value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).
- u16FreeGroupIdEnd is the end value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).
- u64InitiatorIEEEAddr is the IEEE (MAC) address of the local node (network initiator)
- u16InitiatorNwkAddr is the network (short) address of the local node (network initiator)

Parent topic: Structures

tsCLD_ZllCommission_NetworkStartRspCommandPayload This structure is used to hold the payload data for a Touchlink Network Start Response command.

```
typedef struct
{
    uint32   u32TransactionId;
    uint8    u8Status;
    uint64   u64ExtPanId;
    uint8    u8NwkUpdateId;
    uint8    u8LogicalChannel;
    uint16   u16PanId;
```

(continues on next page)

(continued from previous page)

```
} tsCLD_ZllCommission_NetworkStartRspCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- u8Status indicates the outcome of the corresponding Network Start Request: 0x00 for success, 0x01 for failure.
- u64ExtPanId is the Extended PAN ID (EPID) of the new network (this will be the value specified in the corresponding request or a value chosen by the local node).
- u8NwkUpdateId is the current value of the Network Update Identifier, which will be set to zero for a new network (see Section 44.4.3).
- u8LogicalChannel is the number of the IEEE 802.15.4 radio channel to be used by the network (this will be the value specified in the corresponding request or a value chosen by the local node).
- u16PanId is the PAN ID of the new network (this will be the value specified in the corresponding request or a value chosen by the local node).

Parent topic: Structures

tsCLD_ZllCommission_NetworkJoinRouterReqCommandPayload This structure is used to hold the payload data for a Touchlink Network Join Router Request command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint64  u64ExtPanId;
    uint8   u8KeyIndex;
    uint8   au8NwkKey[16];
    uint8   u8NwkUpdateId;
    uint8   u8LogicalChannel;
    uint16  u16PanId;
    uint16  u16NwkAddr;
    uint16  u16GroupIdBegin;
    uint16  u16GroupIdEnd;
    uint16  u16FreeNwkAddrBegin;
    uint16  u16FreeNwkAddrEnd;
    uint16  u16FreeGroupIdBegin;
    uint16  u16FreeGroupIdEnd;
} tsCLD_ZllCommission_NetworkJoinRouterReqCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.
- u64ExtPanId is the Extended PAN ID (EPID) of the network.
- u8KeyIndex is a value indicating the type of security key used to encrypt the network key in au8NwkKey. The valid values are as follows (all other values are reserved for future use):
  - 0: Development key, used during development before ZigBee certification
  - 4: Master key, used after successful ZigBee certification
  - 15: Certification key, used during ZigBee certification testing
- au8NwkKey[16] is the 128-bit network key encrypted using the key specified in u8KeyIndex.

- u8NwkUpdateId is the current value of the Network Update Identifier. This identifier takes a value in the range 0x00 to 0xFF and is incremented when a network update has occurred which requires the network settings on the nodes to be changed.
- u8LogicalChannel is the number of the IEEE 802.15.4 radio channel used by the network.
- u16PanId is the PAN ID of the network
- u16NwkAddr is the 16-bit network (short) address assigned to the target node
- u16GroupIdBegin is the start value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- u16GroupIdEnd is the end value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- u16FreeNwkAddrBegin is the start address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- u16FreeNwkAddrEnd is the end address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- u16FreeGroupIdBegin is the start value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).
- u16FreeGroupIdEnd is the end value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).

Parent topic: Structures

tsCLD_ZllCommission_NetworkJoinRouterRspCommandPayload This structure is used to hold the payload data for a Touchlink Network Join Router Response command.

```
typedef struct
{
    uint32   u32TransactionId;
    uint8    u8Status;
} tsCLD_ZllCommission_NetworkJoinRouterRspCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- u8Status indicates the outcome of the corresponding Network Join Router Request: 0x00 for success, 0x01 for failure.

Parent topic: Structures

tsCLD_ZllCommission_NetworkJoinEndDeviceReqCommandPayload This structure is used to hold the payload data for a Touchlink Network Join End Device Request command.

```
typedef struct
{
    uint32   u32TransactionId;
    uint64   u64ExtPanId;
    uint8    u8KeyIndex;
    uint8    au8NwkKey[16];
```

(continues on next page)

(continued from previous page)

```
    uint8   u8NwkUpdateId;
    uint8   u8LogicalChannel;
    uint16  u16PanId;
    uint16  u16NwkAddr;
    uint16  u16GroupIdBegin;
    uint16  u16GroupIdEnd;
    uint16  u16FreeNwkAddrBegin;
    uint16  u16FreeNwkAddrEnd;
    uint16  u16FreeGroupIdBegin;
    uint16  u16FreeGroupIdEnd;
} tsCLD_ZllCommission_NetworkJoinEndDeviceReqCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.
- u64ExtPanId is the Extended PAN ID (EPID) of the network.
- u8KeyIndex is a value indicating the type of security key used to encrypt the network key in au8NwkKey. The valid values are as follows (all other values are reserved for future use):
  - 0: Development key, used during development before ZigBee certification
  - 4: Master key, used after successful ZigBee certification
  - 15: Certification key, used during ZigBee certification testing
- au8NwkKey[16] is the 128-bit network key encrypted using the key specified in u8KeyIndex.
- u8NwkUpdateId is the current value of the Network Update Identifier. This identifier takes a value in the range 0x00 to 0xFF and is incremented when a network update has occurred which requires the network settings on the nodes to be changed.
- u8LogicalChannel is the number of the IEEE 802.15.4 radio channel used by the network.
- u16PanId is the PAN ID of the network.
- u16NwkAddr is the 16-bit network (short) address assigned to the target node.
- u16GroupIdBegin is the start value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- u16GroupIdEnd is the end value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- u16FreeNwkAddrBegin is the start address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- u16FreeNwkAddrEnd is the end address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- u16FreeGroupIdBegin is the start value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).
- u16FreeGroupIdEnd is the end value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).

Parent topic: Structures

tsCLD_ZllCommission_NetworkJoinEndDeviceRspCommandPayload This structure is used to hold the payload data for a Touchlink Network Join End Device Response command.

```
typedef struct
{
    uint32 u32TransactionId;
    uint8  u8Status;
} tsCLD_ZllCommission_NetworkJoinEndDeviceRspCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- u8Status indicates the outcome of the corresponding Network Join End Device Request: 0x00 for success, 0x01 for failure.

Parent topic: Structures

tsCLD_ZllCommission_NetworkUpdateReqCommandPayload This structure is used to hold the payload data for a Touchlink Network Update Request command.

```
typedef struct
{
 uint32  u32TransactionId;
 uint64  u64ExtPanId;
 uint8u8NwkUpdateId;
 uint8u8LogicalChannel;
 uint16  u16PanId;
 uint16  u16NwkAddr;
} tsCLD_ZllCommission_NetworkUpdateReqCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.
- u64ExtPanId is the Extended PAN ID (EPID) of the network.
- u8NwkUpdateId is the current value of the Network Update Identifier (see Section 44.4.3).
- u8LogicalChannel is the number of the IEEE 802.15.4 radio channel used by the network.
- u16PanId is the PAN ID of the network.
- u16NwkAddr is the 16-bit network (short) address assigned to the target node.

Parent topic: Structures

tsCLD_ZllUtilityCustomDataStructure This structure is used to hold custom data for a Commissioning Utility instance of the Touchlink Commissioning cluster.

```
typedef struct
{
 tsZCL_ReceiveEventAddress sRxAddr;
 tsZCL_CallBackEvent sCustomCallBackEvent;
 tsCLD_ZllUtilityCallBackMessage sCallBackMessage;
} tsCLD_ZllUtilityCustomDataStructure;
```

where:

- sRxAddr is a ZCL structure containing the destination address of the command.
- sCustomCallBackEvent is the ZCL event structure for the command.
- sCallBackMessage is a structure containing the command ID and payload (see Section 44.8.19).

Parent topic: Structures

tsCLD_ZllUtilityCallBackMessage This structure contains the command ID and payload for a received Commissioning Utility command.

```
typedef struct
{
 uint8u8CommandId;
 union
 {
  tsCLD_ZllUtility_EndpointInformationCommandPayload
 *psEndpointInfoPayload;
  tsCLD_ZllUtility_GetGroupIdReqCommandPayload
 *psGetGroupIdReqPayload;
  tsCLD_ZllUtility_GetGroupIdRspCommandPayload
 *psGetGroupIdRspPayload;
  tsCLD_ZllUtility_GetEndpointListReqCommandPayload
 *psGetEndpointlistReqPayload;
  tsCLD_ZllUtility_GetEndpointListRspCommandPayload
 *psGetEndpointListRspPayload;
 } uMessage;
} tsCLD_ZllUtilityCallBackMessage;
```

where:

- u8CommandId is the command ID - enumerations are provided, as detailed in Section 44.6.2.
- uMessage contains the payload of the command, where the structure used depends on the command ID (the structures are detailed in the sections below).

Parent topic: Structures

tsCLD_ZllUtility_EndpointInformationCommandPayload This structure is used to hold the payload data for a Commissioning Utility Endpoint Information command.

```
typedef struct
{
 uint64   u64IEEEAddr;
 uint16   u16NwkAddr;
 uint8u8Endpoint;
 uint16   u16ProfileID;
 uint16   u16DeviceID;
 uint8u8Version;
} tsCLD_ZllUtility_EndpointInformationCommandPayload;
```

where:

- u64IEEEAddr is the IEEE (MAC) address of the local node.
- u16NwkAddr is the network (short) address of the local node.
- u8Endpoint is the number of the local endpoint (1-240).
- u16ProfileID is the identifier of the ZigBee application profile supported on the local endpoint.

- u16DeviceID is identifier of the ZigBee device on the local endpoint.
- u8Version specifies the version number of the ZigBee device on the local endpoint.

Parent topic: Structures

Parent topic:*Touchlink Commissioning Cluster*

Enumerations

Touchlink event enumerations The event types generated by the Touchlink part of the Touchlink Commissioning cluster are enumerated in the teCLD_ZllCommission_Command structure below:

```
typedef enum PACK
{
  E_CLD_COMMISSION_CMD_SCAN_REQ  0x00,
  E_CLD_COMMISSION_CMD_SCAN_RSP,
  E_CLD_COMMISSION_CMD_DEVICE_INFO_REQ,
  E_CLD_COMMISSION_CMD_DEVICE_INFO_RSP,
  E_CLD_COMMISSION_CMD_IDENTIFY_REQ  0x06,
  E_CLD_COMMISSION_CMD_FACTORY_RESET_REQ,
  E_CLD_COMMISSION_CMD_NETWORK_START_REQ  0x10,
  E_CLD_COMMISSION_CMD_NETWORK_START_RSP,
  E_CLD_COMMISSION_CMD_NETWORK_JOIN_ROUTER_REQ,
  E_CLD_COMMISSION_CMD_NETWORK_JOIN_ROUTER_RSP,
  E_CLD_COMMISSION_CMD_NETWORK_JOIN_END_DEVICE_REQ,
  E_CLD_COMMISSION_CMD_NETWORK_JOIN_END_DEVICE_RSP,
  E_CLD_COMMISSION_CMD_NETWORK_UPDATE_REQ,
} teCLD_ZllCommission_Command;
```

Parent topic: Enumerations

Commissioning utility event enumerations The event types generated by the Commissioning Utility part of the Touchlink Commissioning cluster are enumerated in the teCLD_ZllUtility_Command structure below:

```
typedef enum PACK
{
 E_CLD_UTILITY_CMD_ENDPOINT_INFO = 0x40,
 E_CLD_UTILITY_CMD_GET_GROUP_ID_REQ_RSP,
 E_CLD_UTILITY_CMD_GET_ENDPOINT_LIST_REQ_RSP,
} teCLD_ZllUtility_Command;
```

Parent topic: Enumerations

Parent topic:*Touchlink Commissioning Cluster*

Compile-time options This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the Touchlink Commissioning cluster.

The Touchlink Commissioning cluster is enabled as follows:

- **Touchlink -** To enable the cluster, define CLD_ZLL_COMMISSION, then:
  - to enable the cluster as a server, define ZLL_COMMISSION_SERVER
  - to enable the cluster as a client, define ZLL_COMMISSION_CLIENT

- **Commissioning Utility -** To enable the cluster, define CLD_ZLL_UTILITY, then:
  - to enable the cluster as a server, define ZLL_UTILITY_SERVER
  - to enable the cluster as a client, define ZLL_UTILITY_CLIENT

Parent topic:*Touchlink Commissioning Cluster*

Part XI: Appliances Clusters This part comprises four chapters:

- *Chapter 45* details the **Appliance Control** cluster
- *Chapter 46* details the **Appliance Identification** cluster
- *Chapter 47* details the **Appliance Events and Alerts** cluster
- *Chapter 48* details the **Appliance Statistics** cluster

Appliance Control Cluster This chapter outlines the Appliance Control cluster which provides an interface for remotely controlling appliances in the home.

The Appliance Control cluster has a Cluster ID of 0x001B.

Overview The Appliance Control cluster provides an interface for the remote control and programming of home appliances (e.g. a washing machine) by sending basic operational commands such as start, pause, stop.

The cluster is enabled by defining CLD_APPLIANCE_CONTROL in the **zcl_options.h** file. Further compile-time options for the Appliance Control cluster are detailed in Section 45.10.

All attributes of the Appliance Control cluster are in the 'Appliance Functions' attribute set.

Parent topic:*Appliance Control Cluster*

Cluster structure and attributes The structure definition for the Appliance Control cluster (server) is:

```
typedef struct
{
#ifdef APPLIANCE_CONTROL_SERVER
    zuint16                     u16StartTime;
    zuint16                     u16FinishTime;
#ifdef CLD_APPLIANCE_CONTROL_REMAINING_TIME
    zuint16                     u16RemainingTime;
#endif
#ifdef CLD_APPLIANCE_CONTROL_ATTRIBUTE_REPORTING_STATUS
    zenum8                       u8AttributeReportingStatus;
#endif
#endif
    zuint16                     u16ClusterRevision;
} tsCLD_ApplianceControl;
```

where:

- u16StartTime is a bitmap representing the start-time of a 'running' cycle of the appliance, as follows:

Bits	Description
0-5	Minutes part of the start-time, in the range 0 to 59

(may be absolute or relative time - see below)

| |6-7|Type of time encoding:

- 0x0: Relative time - start-time is a delay from the time that the attribute was set
- 0x1: Absolute time - start-time is an actual time of the 24-hour clock
- 0x2-0x3: Reserved

The defaults are absolute time for ovens and relative time for other appliances.

| |8-15|Hours part of the start-time:

- in the range 0 to 255, if relative time selected
- in the range 0 to 23, if absolute time selected

|

- u16FinishTime is a bitmap representing the stop-time of a 'running' cycle of the appliance, as follows:

Bits	Description
0-5	Minutes part of the stop-time, in the range 0 to 59

(may be absolute or relative time - see below)

| |6-7|Type of time encoding:

- 0x0: Relative time - stop-time is a delay from the time that the attribute was set
- 0x1: Absolute time - stop-time is an actual time of the 24-hour clock
- 0x2-0x3: Reserved

The defaults are absolute time for ovens and relative time for other appliances.

| |8-15|Hours part of the stop-time:

- in the range 0 to 255, if relative time selected
- in the range 0 to 23, if absolute time selected

|

- u16RemainingTime is an optional attribute indicating the time, in minutes, remaining in the current 'running' cycle of the appliance (time until the end of the cycle) - this attribute is constantly updated during the running cycle and is zero when the appliance is not running
- u8AttributeReportingStatus is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Appliance Control Cluster*

Attributes for default reporting The following attributes of the Appliance Control cluster can be selected for default reporting:

```
u16StartTime
u16FinishTime
```

- u16RemainingTime

Attribute reporting (including default reporting) is described in *Appendix B*. Enabling reports for these attributes is described in Appendix B.3.6.

Parent topic:*Appliance Control Cluster*

Sending commands The Appliance Control cluster server resides on the appliance to be controlled (e.g. a washing machine) and the cluster client resides on the controlling device (normally a remote control unit).

The commands from the client to the server can be of two types:

- 'Execution' commands, requesting appliance operations
- 'Status' commands, requesting appliance status information

In addition, status notification messages can be sent unsolicited from the server to the client.

Sending the above messages is described in the sub-sections below.

Execution Commands from Client to Server An 'execution' command can be sent from the client to request that an operation is performed on the appliance (server) - the request is sent in an 'Execution of Command' message. The application on the client can send this message by calling the function **eCLD_ACExecutionOfCommandSend()**.

The possible operations depend on the target appliance but the following operations are available to be specified in the message payload (described in Section 45.9.2):

- Start appliance cycle
- Stop appliance cycle
- Pause appliance cycle
- Start superfreezing cycle
- Stop superfreezing cycle
- Start supercooling cycle
- Stop supercooling cycle
- Disable gas
- Enable gas

In the start and stop commands, the start-time and end-time can be specified. The commands are fully detailed in the British Standards document BS EN 50523.

The application on the server (appliance) will be notified of the received command by an E_CLD_APPLIANCE_CONTROL_CMD_EXECUTION_OF_COMMAND event (Appliance Control events are described in Section 45.5). The required command is specified in the payload of the message, which is contained in the above event. The application must then perform the requested command (if possible).

Parent topic: Sending commands

Status Commands from Client to Server The application on the cluster client can request the current status of the appliance by sending a 'Signal State' message to the cluster server on the appliance. This message can be sent by calling the function **eCLD_ACSignalStateSend()**. This function returns immediately and the requested status information is later returned in an E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE event, which is generated when a response arrives from the server (Appliance Control events are described in Section 45.5).

Note: The cluster server handles the 'Signal State' message automatically and returns the requested status information in a 'Signal State Response' message to the client.

The appliance status information from the message payload is contained in the above event - for details of this payload and the status information, refer to Section 45.9.3.

Parent topic: Sending commands

Status Notifications from Server to Client The cluster server on the appliance can send unsolicited status notifications to the client in 'Signal State Notification' messages. A message of this kind can be sent by the application on the server by calling either of the following functions:

- **eCLD_ACSignalStateNotificationSend()**
- **eCLD_ACSignalStateResponseORSignalStateNotificationSend()**

Note: The latter function is also used internally by the cluster server to send a 'Signal State Response' message - see Section 45.4.2.

The appliance status information from the 'Signal State Notification' message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION, which is generated when the notification arrives from the server (Appliance Control events are described in Section 45.5). The appliance status information from the message payload is contained in the above event - for details of this payload and the status information, refer to Section 45.9.3.

Parent topic: Sending commands

Parent topic:*Appliance Control Cluster*

Appliance control events The Appliance Control cluster has its own events that are handled through the callback mechanism described in *Chapter 3*. The cluster contains its own event handler. If a device uses this cluster then application-specific Appliance Control event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function will then be invoked when an Appliance Control event occurs and needs the attention of the application.

For an Appliance Control event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_ApplianceControlCallBackMessage structure:

```
typedef struct
{
    uint8   u8CommandId;
    bool    *pbApplianceStatusTwoPresent;
    union
    {
        tsCLD_AC_ExecutionOfCommandPayload *psExecutionOfCommandPayload;
        tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload
                                    *psSignalStateResponseAndNotificationPayload;
    } uMessage;
} tsCLD_ApplianceControlCallBackMessage;
```

When an Appliance Control event occurs, one of four command types could have been received. The relevant command type is specified through the u8CommandId field of the tsSM_CallBackMessage structure. The possible command types are detailed the tables below for events generated on a server and a client.

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_CONTROL_	

CMD_EXECUTION_OF_COMMAND

|An 'Execution of Command' message has been received by the server (appliance), requesting an opera-tion on the appliance| |E_CLD_APPLIANCE_CONTROL_

CMD_SIGNAL_STATE

|A 'Signal State' message has been received by the server (appliance), requesting the status of the appli-ance|

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE	A response to a 'Signal State' message has been received by the client, containing the requested appli-ance status
E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION	A 'Signal State' notification message has been received by the client, containing unsolicited status information

Parent topic:*Appliance Control Cluster*

Functions The following Appliance Control cluster functions are provided:

1. eCLD_ApplianceControlCreateApplianceControl
2. eCLD_ACExecutionOfCommandSend
3. eCLD_ACSignalStateSend
4. eCLD_ACSignalStateResponseORSignalStateNotificationSend
5. eCLD_ACSignalStateNotificationSend
6. eCLD_ACChangeAttributeTime

eCLD_ApplianceControlCreateApplianceControl

```
teZCL_Status eCLD_ApplianceControlCreateApplianceControl(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits,
    tsCLD_ApplianceControlCustomDataStructure
    *psCustomDataStructure);
```

Description This function creates an instance of the Appliance Control cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device).

This function will create an Appliance Control cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to *Appendix D.*

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in the *ZigBee Devices User Guide**(JNUG3131)*.

When used, this function must be the first Appliance Control cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length is automatically adjusted by the compiler using the following declaration:

```
uint8 au8ApplianceControlAttributeControlBits
[(sizeof(asCLD_ApplianceControlClusterAttributeDefinitions) / sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Appliance Control cluster. This parameter can refer to a pre-filled structure called sCLD_ApplianceControl which is provided in the **ApplianceControl.h** file.
- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_ApplianceControl which defines the attributes of Appliance Control cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.
- *psCustomDataStructure* Pointer to a structure containing the storage for internal functions of the cluster (see Section 45.9.4)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_ACExecutionOfCommandSend

```
teZCL_Status eCLD_ACExecutionOfCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_AC_ExecutionOfCommandPayload *psPayload);
```

Description This function can be used on an Appliance Control cluster client to send an 'Execution of Command' message to a cluster server (appliance), where this message may specify one of the following control commands:

- Start appliance cycle
- Stop appliance cycle
- Pause appliance cycle
- Start superfreezing cycle
- Stop superfreezing cycle
- Start supercooling cycle
- Stop supercooling cycle
- Disable gas
- Enable gas

The required command is specified in the payload of the message (a pointer to this payload must be provided). The commands are fully detailed in the British Standards document BS EN 50523.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameter

- *u8SourceEndPointId* Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload* Pointer to a structure containing the payload for the message (see Section 45.9.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL

- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_ACSignalStateSend

```
teZCL_Status eCLD_ACSignalStateSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on an Appliance Control cluster client to send a 'Signal State' message to a cluster server (appliance), which requests the status of the appliance. The function returns immediately and the requested status information is later returned in the following event, which is generated when a response is received from the server:

E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the message is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_ACSignalStateResponseORSignalStateNotificationSend

```
teZCL_Status eCLD_ACSignalStateResponseORSignalStateNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
```

(continues on next page)

(continued from previous page)

```
    teCLD_ApplianceControl_ServerCommandId eCommandId,
    bool bApplianceStatusTwoPresent,
    tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload
*psPayload);
```

Description This function can be used on an Appliance Control cluster server to send a 'Signal State Response' message (in reply to a 'Signal State Request' message) or an unsolicited 'Signal State Notification' message to a cluster client.

The command to be sent must be specified as one of:

- E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE
- E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *eCommandId* Enumeration indicating the command to be sent (see above and Section 45.8.3)
- *bApplianceStatusTwoPresent* Boolean indicating whether additional appliance status data is present in payload:
  - TRUE - Present
  - FALSE - Not present
- *psPayload* Pointer to structure containing payload for message (see above and Section 45.9.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_ACSignalStateNotificationSend

```
teZCL_Status eCLD_ACSignalStateNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    bool bApplianceStatusTwoPresent,
    tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload**
*psPayload);
```

Description This function can be used on an Appliance Control cluster server to send an unsolicited 'Signal State Notification' message to a cluster client. The function is an alternative to **eCLD_ACSignalStateResponseORSignalStateNotificationSend()**.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the message is sent
- *psDestinationAddress*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *bApplianceStatusTwoPresent*: Boolean indicating whether additional appliance status data is present in payload:
  - TRUE - Present
  - FALSE - Not present
- **psPPayload*: Pointer to structure containing payload for message (see above and Section 45.9.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_ACChangeAttributeTime

```
teZCL_Status eCLD_ACChangeAttributeTime(
        uint8 u8SourceEndPointId,
        teCLD_ApplianceControl_Cluster_AttrID eAttributeTimeId,
        uint16 u16TimeValue);
```

Description This function can be used on an Appliance Control cluster server (appliance) to update the time attributes of the cluster (start time, finish time, remaining time). This is particularly useful if the host node has its own timer.

The target attribute must be specified using one of:

- E_CLD_APPLIANCE_CONTROL_ATTR_ID_START_TIME
- E_CLD_APPLIANCE_CONTROL_ATTR_ID_FINISH_TIME
- E_CLD_APPLIANCE_CONTROL_ATTR_ID_REMAINING_TIME

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *eAttributeTimeId*: Identifier of attribute to be updated (see above and Section 45.9.1)
- *u16TimeValue*: UTC time to set

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic:*Appliance Control Cluster*

Return codes The Appliance Control cluster functions use the ZCL return codes, listed in Section7.2.

Parent topic:*Appliance Control Cluster*

Enumerations

'Attribute ID' Enumerations The following structure contains the enumerations used to identify the attributes of the Appliance Control cluster.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_CONTROL_ATTR_ID_START_TIME      = 0x0000,
    E_CLD_APPLIANCE_CONTROL_ATTR_ID_FINISH_TIME,
    E_CLD_APPLIANCE_CONTROL_ATTR_ID_REMAINING_TIME
} teCLD_ApplianceControl_Cluster_AttrID;
```

Parent topic: Enumerations

'Client Command ID' Enumerations The following enumerations are used in commands issued on a cluster client.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_CONTROL_CMD_EXECUTION_OF_COMMAND = 0x00,
    E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE,
} teCLD_ApplianceControl_ClientCommandId;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_APPLIANCE_CONTROL_CMD_EXECUTION_OF_COMM	'Execution of Command' message
E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE	'Signal State' message

Parent topic: Enumerations

'Server command ID' enumerations The following enumerations are used in commands issued on a cluster server.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE = 0x00,
    E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION
} teCLD_ApplianceControl_ServerCommandId;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPO	A response to a 'Signal State' request
E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFI	A 'Signal State' notification

Parent topic: Enumerations

Parent topic:*Appliance Control Cluster*

Structures

tsCLD_ApplianceControlCallBackMessage For an Appliance Control event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_ApplianceControlCallBackMessage structure:

```
typedef struct
{
    uint8    u8CommandId;
```

(continues on next page)

(continued from previous page)

```
    bool    *pbApplianceStatusTwoPresent;
    union
    {
        tsCLD_AC_ExecutionOfCommandPayload                    *psExecutionOfCommandPayload;
        tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload
                                    *psSignalStateResponseAndNotificationPayload;
    } uMessage;
} tsCLD_ApplianceControlCallBackMessage;
```

where:

- u8CommandId indicates the type of Appliance Control command that has been received, one of:
  - E_CLD_APPLIANCE_CONTROL_CMD_EXECUTION_OF_COMMAND
  - E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE
  - E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE
  - E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION
- pbApplianceStatusTwoPresent is a pointer to a boolean indicating whether a second set of non-standard or proprietary status data is available:
  - TRUE - additional status data available
  - FALSE - additional status data unavailable
- uMessage is a union containing the command payload as one of (depending on the value of u8CommandId):
- psExecutionOfCommandPayload is a pointer to the payload of an 'Execution of Command' message (see Section 45.9.2)
- psSignalStateResponseAndNotificationPayload is a pointer to the payload of a 'Signal State' response or notification message (see Section 45.9.3)

Parent topic: Structures

tsCLD_AC_ExecutionOfCommandPayload This structure contains the payload for an "Execution of Command" message.

```
typedef struct
{
        zenum8              eExecutionCommandId;
} stsCLD_AC_ExecutionOfCommandPayload;
```

where eExecutionCommandId is a value representing the command to be executed - the commands are detailed in the British Standards document BS EN 50523.

Parent topic: Structures

tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload This structure contains the payload for a "Signal State" response or notification message.

```
typedef struct
{
        zenum8   eApplianceStatus;
        zuint8              u8RemoteEnableFlagAndDeviceStatus;
        zuint24                     u24ApplianceStatusTwo;
```

(continues on next page)

(continued from previous page)

```
} tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload;
```

where:

- eApplianceStatus is a value indicating the reported appliance status (the relevant status values depend on the appliance):

Status Value	Description
0x00	Reserved
0x01	Appliance in off state
0x02	Appliance in stand-by
0x03	Appliance already programmed
0x04	Appliance already programmed and ready to start
0x05	Appliance is running
0x06	Appliance is in pause state
0x07	Appliance end programmed tasks
0x08	Appliance is in a failure state
0x09	Appliance programmed tasks have been interrupted
0x0A	Appliance in idle state
0x0B	Appliance rinse hold
0x0C	Appliance in service state
0x0D	Appliance in superfreezing state
0x0E	Appliance in supercooling state
0x0F	Appliance in superheating state
0x10-0x3F	Reserved
0x40-0x7F	Non-standardised
0x80-0xFF	Proprietary

- u8RemoteEnableFlagAndDeviceStatus is a bitmap value indicating the status of the relationship between the appliance and the remote control unit as well as the type of additional status information reported in u24ApplianceStatusTwo:

Bits	Field	Values/Description
0-3	Remote Enable Flags	Status of remote control link:

- 0x0: Disabled
- 0x1: Enabled remote and energy control
- 0x2-0x06: Reserved
- 0x7: Temporarily locked/disabled
- 0x8-0xE: Reserved
- 0xF: Enabled remote control

| |4-7|Device Status 2|Type of information in u24ApplianceStatusTwo:

- 0x0: Proprietary
- 0x1: Proprietary
- 0x2: IRIS symptom code
- 0x3-0xF: Reserved

|

- u24ApplianceStatusTwo is a value indicating non-standard or proprietary status information about the appliance. The type of status information represented by this value is indicated

in the 'Device Status 2' field of u8RemoteEnableFlagAndDeviceStatus. In the case of an IRIS symptom code, the three bytes of this value represent a 3-digit code.

Parent topic: Structures

tsCLD_ApplianceControlCustomDataStructure The Appliance Control cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress                    sReceiveEventAddress;
    tsZCL_CallBackEvent                          sCustomCallBackEvent;
    tsCLD_ApplianceControlCallBackMessage        sCallBackMessage;
} tsCLD_ApplianceControlCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Parent topic:*Appliance Control Cluster*

Compile-time options This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the Appliance Control cluster.

To enable the Appliance Control cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_APPLIANCE_CONTROL
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define APPLIANCE_CONTROL_SERVER
#define APPLIANCE_CONTROL_CLIENT
```

The Appliance Control cluster contains macros that may be optionally specified at compile-time by adding one or more of the following lines to the **zcl_options.h** file.

Optional attributes Add this line to enable the optional Time Remaining attribute:

```
#define CLD_APPLIANCE_CONTROL_REMAINING_TIME
```

Global Attributes Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_LEVELCONTROL_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_LEVELCONTROL_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Appliance Control Cluster*

Appliance Identification Cluster This chapter outlines the Appliance Identification cluster, which provides an interface for obtaining and setting basic appliance information.

The Appliance Identification cluster has a Cluster ID of 0x0B00.

- **Overview**
- **Cluster structure and attributes**
- **Functions**
- **Return codes**
- **Enumerations**
- **Compile-time options**

Appliance Events and Alerts Cluster This chapter outlines the Appliance Events and Alerts cluster, which provides an interface for the notification of significant events and alert situations.

The Appliance Events and Alerts cluster has a Cluster ID of 0x0B02.

Overview The Appliance Events and Alerts cluster provides an interface for sending notifications of appliance events (for example, target temperature reached) and alerts (for example, alarms).

The cluster is enabled by defining CLD_APPLIANCE_EVENTS_AND_ALERTS in the **zcl_options.h** file. Further compile-time options for the Appliance Events and Alerts cluster are detailed in Section 47.9.

Events are notified in terms of header and event identifier fields (an event may occur when the appliance reaches a certain state, such as the end of its operational cycle).

Alerts are notified in terms of the following fields:

- Alert identification value
- Alert category, one of: Warning, Danger, Failure
- Presence/recovery flag (indicating alert has been either detected or recovered)

Parent topic:*Appliance Events and Alerts Cluster*

Cluster structure and attribute The structure definition for the Appliance Events and Alerts cluster (server) is.

```
typedef struct
{
 zuint16    u16ClusterRevision;
} tsCLD_ApplianceEventsAndAlerts;
```

where u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Appliance Events and Alerts Cluster*

Sending Messages The Appliance Events and Alerts cluster server resides on the appliance (for example, a washing machine) and the cluster client resides on a controlling device (normally a remote control unit).

Messages can be sent between the client and the server in the following ways:

- Alerts that are active on the appliance can be requested by the client by sending a 'Get Alerts' message to the server (which replies with a 'Get Alerts Response' message).
- Alerts that are active on the appliance can be sent unsolicited from the server to the client in an 'Alerts Notification' message.
- The server can notify the client of an appliance event by sending an unsolicited 'Event Notification' message to the client

Sending the above messages is described in the sub-sections below.

'Get Alerts' Messages from Client to Server The application on the cluster client can request the alerts that are currently active on the appliance by sending a 'Get Alerts' message to the server - this message is sent by calling the function **eCLD_AEAAGetAlertsSend()**. This function returns immediately and the requested alerts are later returned in an E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS event, which is generated when a response arrives from the server (Appliance Events and Alerts events are described in Section 47.4).

Note: The cluster server handles the 'Get Alerts' message automatically and returns the requested alerts in a 'Get Alerts Response' message to the client.

The appliance alerts from the message payload are contained in the above event - for details of this payload and the alert information, refer to Section 47.8.2. Up to 15 alerts can be reported in a single response.

Parent topic: Sending Messages

'Alerts Notification' Messages from Server to Client The cluster server on the appliance can send unsolicited alert notifications to the client in 'Alerts Notification' messages. The application on the server can send a message of this kind by calling either of the following functions:

- **eCLD_AEAAAlertsNotificationSend()**
- **eCLD_AEAAGetAlertsResponseORAlertsNotificationSend()**

Note: The latter function is also used internally by the cluster server to send a 'Get Alerts Response' message - see Section 47.3.1.

The appliance status information from the 'Alerts Notification' message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION, which is generated when the notification arrives from the server (Appliance Events and Alerts events are described in Section 47.4). The appliance alerts from the message payload are contained in the above event - for details of this payload and the alert information, refer to Section 47.8.2. Up to 15 alerts can be reported in a single notification.

Parent topic: Sending Messages

'Event Notification' Messages from Server to Client The cluster server on the appliance can send unsolicited event notifications to the client in 'Event Notification' messages, where each message reports a single appliance event (for example, oven has reached its target temperature). A message of this kind can be sent by the application on the server by calling the function **eCLD_AEAAEventNotificationSend()**.

The appliance event information from the 'Event Notification' message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION, which is generated when the notification arrives from the server (Appliance Events and Alerts events are described in Section 47.4). The appliance event from the message payload is contained in the above client event - for details of this payload and the embedded appliance event information, refer to Section 47.8.3.

Parent topic: Sending Messages

Parent topic:*Appliance Events and Alerts Cluster*

Appliance Events and Alerts Events The Appliance Events and Alerts cluster has its own events that are handled through the callback mechanism described in *Chapter 3*. The cluster contains its own event handler. If a device uses this cluster then application-specific Appliance Events and Alerts event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function is then invoked when an Appliance Events and Alerts event occurs and needs the attention of the application.

For an Appliance Events and Alerts event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_ApplianceEventsAndAlertsCallBackMessage structure:

```
typedef struct
{
  uint8  u8CommandId
  union
  {
  tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload
  *psGetAlertsResponseORAlertsNotificationPayload;
  tsCLD_AEAA_EventNotificationPayload
  *psEventNotificationPayload;
  }uMessage;
} tsCLD_ApplianceEventsAndAlertsCallBackMessage;
```

When an Appliance Events and Alerts event occurs, one of four command types could have been received. The relevant command type is specified through the u8CommandId field of the tsSM_CallBackMessage structure. The possible command types are detailed the tables below for events generated on a server and a client.

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_Cl	A 'Get Alerts' request has been received by the server (appliance)

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_EVENTS_AND_AL	A response to a 'Get Alerts' request has been received by the client, containing the requested alerts (up to 15)
E_CLD_APPLIANCE_EVENTS_AND_AL	An 'Alerts Notification' message has been received by the client, containing unsolicited alerts (up to 15)
E_CLD_APPLIANCE_EVENTS_AND_AL	An 'Event Notification' message has been received by the client

Parent topic:*Appliance Events and Alerts Cluster*

Functions The following Appliance Events and Alerts cluster functions are provided:

1. eCLD_ApplianceEventsAndAlertsCreateApplianceEventsAndAlerts
2. eCLD_AEAAGetAlertsSend
3. eCLD_AEAAGetAlertsResponseORAlertsNotificationSend
4. eCLD_AEAAAlertsNotificationSend
5. eCLD_AEAAEventNotificationSend

eCLD_ApplianceEventsAndAlertsCreateApplianceEventsAndAlerts

```
teZCL_Status eCLD_ApplianceEventsAndAlertsCreateApplianceEventsAndAlerts(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    oid *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits,
    tsCLD_ApplianceEventsAndAlertsCustomDataStructure
*psCustomDataStructure);
```

Description This function creates an instance of the Appliance Events and Alerts cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function creates an Appliance Events and Alerts cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to *Appendix D.*

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in the *ZigBee Devices User Guide**(JNUG3131)*.

When used, this function must be the first Appliance Events and Alerts cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length is automatically adjusted by the compiler using the following declaration:

```
uint8 au8ApplianceEventsAndAlertsAttributeControlBits
    [(sizeof(asCLD_ApplianceEventsAndAlertsClusterAttributeDefinitions)
    / sizeof(tsZCL_AttributeDefinition))];
```

Parameters

Returns **Parent topic:**Functions

eCLD_AEAAGetAlertsSend

```
teZCL_Status eCLD_AEAAGetAlertsSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on an Appliance Events and Alerts cluster client to send a 'Get Alerts' message to a cluster server (appliance).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameter

Returns **Parent topic:**Functions

eCLD_AEAAGetAlertsResponseORAlertsNotificationSend

```
teZCL_Status eCLD_AEAAGetAlertsResponseORAlertsNotificationSend(
uint8 u8SourceEndPointId,
uint8 u8DestinationEndPointId,
tsZCL_Address *psDestinationAddress,
uint8 *pu8TransactionSequenceNumber,
teCLD_ApplianceEventsAndAlerts_CommandId eCommandId,
tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload
*psPayload);
```

Description This function can be used on an Appliance Events and Alerts cluster server to send a 'Get Alerts Response' message (in reply to a 'Get Alerts' message) or an unsolicited 'Alerts Notification' message to a cluster client.

The command to be sent must be specified as one of:

- E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS
- E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

Returns **Parent topic:**Functions

eCLD_AEAAAlertsNotificationSend

```
teZCL_Status eCLD_AEAAAlertsNotificationSend(
uint8 u8SourceEndPointId,
uint8 u8DestinationEndPointId,
tsZCL_Address *psDestinationAddress,
uint8 *pu8TransactionSequenceNumber,
tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload
*psPayload);
```

Description This function can be used on an Appliance Events and Alerts cluster server to send an unsolicited 'Alerts Notification' message to a cluster client. The function is an alternative to **eCLD_AEAAGetAlertsResponseORAlertsNotificationSend()**.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

Returns **Parent topic:**Functions

eCLD_AEAAEventNotificationSend

```
teZCL_Status eCLD_AEAAEventNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_AEAA_EventNotificationPayload *psPayload);
```

Description This function can be used on an Appliance Events and Alerts cluster server (appliance) to send an 'Event Notification' message to a cluster client, to indicate that an incident has occurred.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

Returns **Parent topic:**Functions

Parent topic:*Appliance Events and Alerts Cluster*

Return codes The Appliance Events and Alerts cluster functions use the ZCL return codes, listed in Section 7.2.

Parent topic:*Appliance Events and Alerts Cluster*

Enumerations

'Command ID' Enumerations The following enumerations are used in commands received on a cluster server or client.

```
typedef enum PACK
{
  E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS = 0x00,
  E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION,
  E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_EVENT_NOTIFICATION
} teCLD_ApplianceEventsAndAlerts_CommandId;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET	'Get Alerts' request (on server) or response (on client)
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALE	Alerts notification (on client)
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_EVE	Events notification (on server)

Parent topic: Enumerations

Parent topic:*Appliance Events and Alerts Cluster*

Structures

tsCLD_ApplianceEventsAndAlertsCallBackMessage For an Appliance Events and Alerts event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_ApplianceEventsAndAlertsCallBackMessage structure:

```
typedef struct
{
 uint8     u8CommandId
 union
 {
 tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload
        *psGetAlertsResponseORAlertsNotificationPayload;
 tsCLD_AEAA_EventNotificationPayload
        *psEventNotificationPayload;
 } uMessage;
} tsCLD_ApplianceEventsAndAlertsCallBackMessage;
```

where:

- u``8CommandId indicates the type of Appliance Events and Alerts command that has been received, one of:
  - E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS
  - E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION
  - E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_EVENT_NOTIFICATION
- u``Message is a union containing the command payload as one of (depending on the value of u8CommandId):

- psGetAlertsResponseORAlertsNotificationPayload is a pointer to the payload of an "Get Alerts" response message or an alerts notification message (see Section 47.8.2)
- psEventNotificationPayload is a pointer to the payload of an events notification message (see Section 47.8.3)

Parent topic: Structures

tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload This structure contains the payload for a 'Get Alerts Response' message or an 'Alerts Notification' message.

```
typedef struct
{
  zuint8  u8AlertsCount;
  zuint24 au24AlertStructure[
            CLD_APPLIANCE_EVENTS_AND_ALERTS_MAXIMUM_NUM_OF_ALERTS];
} tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload;
```

where:

- u8AlertsCount is an 8-bit bitmap containing the following alerts information:

Bits	Description
0-3	Number of reported alerts
4-7	Type of alert:

- 0x0: Unstructured
- 0x1-0xF: Reserved

|

- au24AlertStructure[``] is an array of 24-bit bitmaps, with one bitmap for each reported alert, containing the following information:

Bits	Description
0-7	Alert ID:

- 0x0: Reserved
- 0x01-0x3F: Standardized
- 0x40-0x7F: Non-standardized
- 0x80-0xFF: Proprietary

| |8-11|Category:

- 0x0: Reserved
- 0x1: Warning
- 0x2: Danger
- 0x3: Failure
- 0x4–0xF: Reserved

| |12-13|Presence or recovery:

- 0x0: Presence (alert detected)
- 0x1: Recovery (alert recovered)
- 0x2–0x3: Reserved

| |14-15|Reserved (set to 0x0)| |16-23|Non-standardized or proprietary|

Parent topic: Structures

tsCLD_AEAA_EventNotificationPayload This structure contains the payload for an 'Event Notification' message.

```
typedef struct
{
  zuint8          u8EventHeader;
  zuint8          u8EventIdentification;
} tsCLD_AEAA_EventNotificationPayload;
```

where:

- u8EventHeader is reserved and set to 0
- u8EventIdentification is the identifier of the event being notified:
  - 0x01: End of operational cycle reached
  - 0x02: Reserved
  - 0x03: Reserved
  - 0x04: Target temperature reached
  - 0x05: End of cooking process reached
  - 0x06: Switching off
  - 0xF7: Wrong data
    - (Values 0x00 to 0x3F are standardised, 0x40 to 0x7F are non-standardised, and 0x80 to 0xFF except 0xF7 are proprietary)

Parent topic: Structures

tsCLD_ApplianceEventsAndAlertsCustomDataStructure The Appliance Events and Alerts cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
   tsZCL_ReceiveEventAddress sReceiveEventAddress;
   tsZCL_CallBackEvent sCustomCallBackEvent;
   tsCLD_ApplianceEventsAndAlertsCallBackMessage sCallBackMessage;
} tsCLD_ApplianceEventsAndAlertsCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Parent topic:*Appliance Events and Alerts Cluster*

Compile-time options This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the Appliance Events and Alerts cluster.

To enable the Appliance Events and Alerts cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_APPLIANCE_EVENTS_AND_ALERTS
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define APPLIANCE_EVENTS_AND_ALERTS_SERVER
#define APPLIANCE_EVENTS_AND_ALERTS_CLIENT
```

The Appliance Identification cluster contains macros that may be optionally specified at compile-time by adding one or more of the following lines to the **zcl_options.h** file.

Maximum Number of Alerts Reported The maximum number of alerts that can be reported in a response or notification can be defined (as n) using the following definition in the **zcl_options.h** file:

```
#define CLD_APPLIANCE_EVENTS_AND_ALERTS_MAXIMUM_NUM_OF_ALERTS n
```

The default value is 16, which is the upper limit on this value, and n must therefore not be greater than 16.

Global Attributes Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_APPLIANCE_EVENTS_AND_ALERTS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Parent topic:*Appliance Events and Alerts Cluster*

Appliance Statistics Cluster This chapter outlines the Appliance Statistics cluster, which provides an interface for supplying statistical information about an appliance.

The Appliance Statistics cluster has a Cluster ID of 0x0B03.

Overview The Appliance Statistics cluster provides an interface for sending appliance statistics in the form of data logs to a collector node, which may be a gateway.

The cluster is enabled by defining CLD_APPLIANCE_STATISTICS in the **zcl_options.h** file. Further compile-time options for the Appliance Statistics cluster are detailed in Section 48.10.

The cluster client may obtain logs from the server (appliance) in any of the following ways:

- Unsolicited log notifications sent by the server
- Solicited responses obtained by:
  - Client sending 'Log Queue Request' to enquire whether logs are available
  - Client sending 'Log Request' for each log available
- Semi-solicited responses obtained by:
  - Server sending 'Statistics Available' notification to indicate that logs are available
  - Client sending 'Log Request' for each log available

Parent topic:*Appliance Statistics Cluster*

Cluster structure and attributes The structure definition for the Appliance Statistics cluster (server) is:

```
typedef struct
{
#ifdef APPLIANCE_STATISTICS_SERVER
    zuint32                    u32LogMaxSize;
    zuint8                     u8LogQueueMaxSize;
#endif
    zuint16                    u16ClusterRevision;
} tsCLD_ApplianceStatistics;
```

where:

- u32LogMaxSize is a mandatory attribute which specifies the maximum size, in bytes, of the payload of a log notification and log response. This value should not be greater than 70 bytes (otherwise the Partition cluster is needed)
- u8LogQueueMaxSize is a mandatory attribute which specifies the maximum number of logs in the queue on the cluster server that are available to be requested by the client
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Parent topic:*Appliance Statistics Cluster*

Sending messages The Appliance Statistics cluster server resides on the appliance (e.g. a washing machine) and the cluster client resides on a controlling device (normally a remote control unit).

Messages can be sent between the client and the server in the following ways:

- The client can enquire whether any data logs are available on the appliance (server) by sending a 'Log Queue Request' to the server (which will reply with a 'Log Queue Response' message)
- The server can notify the client that data logs are available by sending an unsolicited 'Statistics Available' message to the client
- The client can request a current data log from the appliance (server) by sending a 'Log Request' message to the server (which will reply with a 'Log Response' message)
- The server can send an unsolicited data log to the client in a 'Log Notification' message

Sending the above messages is described in the sub-sections below.

'Log Queue Request' messages from client to server The application on the cluster client can enquire about the availability of data logs on the appliance by sending a 'Log Queue Request' message to the server. This message is sent by calling the function **eCLD_ASCLogQueueRequestSend()**. This function returns immediately and the log availability is later returned in an E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE event, which is generated when a response arrives from the server (Appliance Statistics events are described in Section 48.5).

Note: The cluster server handles the 'Log Queue Request' message automatically and returns the requested information in a 'Log Queue Response' message to the client.

The log availability information from the message payload is contained in the above event, and comprises the number of logs currently in the log queue and their log IDs - for details of this payload and the availability information, refer to Section 48.9.4.

Parent topic: Sending messages

'Statistics Available' messages from server to client The cluster server can notify the client when data logs are available by sending an unsolicited 'Statistics Available' message to the client. This message contains the number of logs in the log queue and the log IDs. A message of this kind can be sent by the application on the server by calling either of the following functions:

- **eCLD_ASCStatisticsAvailableSend()**
- **eCLD_ASCLogQueueResponseORStatisticsAvailableSend()**

Note:

1. The latter function is also used internally by the cluster server to send a 'Log Queue Response' message - see Section 48.3.1.
2. Before calling either function, the relevant log(s) should be added to the local log queue as described in Section 48.4.1. This is because the logs need to be in the queue to allow the server to perform further actions on them - for example, to process a 'Log Request'.

The log availability information from the 'Statistics Available' message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE, which is generated when the message arrives from the server (Appliance Statistics events are described in Section 48.5). The availability information from the message payload is contained in the above event - for details of this payload and the availability information, refer to Section 48.9.4.

Parent topic: Sending messages

'Log Request' messages from client to server The application on the cluster client can request the log with a particular log ID from the appliance by sending a 'Log Request' message to the server. This message is sent by calling the function **eCLD_ASCLogRequestSend()**. This function returns immediately and the requested log information is later returned in an E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE event, which is generated when a response arrives from the server (Appliance Statistics events are described in Section 48.5).

Note:

1. This function should normally be called after a 'Log Queue Response' or 'Statistics Available' message has been received by the client, indicating that logs are available on the server.
2. The cluster server handles the 'Log Request' message automatically and returns the requested log information in a 'Log Response' message to the client.

The log information from the message payload is contained in the above event - for details of this payload and the supplied log information, refer to Section 48.9.3.

Parent topic: Sending messages

'Log Notification' messages from server to client The cluster server can supply the client with an individual data log by sending an unsolicited 'Log Notification' message to the client. This message is sent by the application on the server by calling either of the following functions:

- **eCLD_ASCLogNotificationSend()**
- **eCLD_ASCLogNotificationORLogResponseSend()**

Note:

1. The latter function is also used internally by the cluster server to send a 'Log Response' message - see Section 48.3.1.

2. Before calling either function, the relevant log should be in the local log queue (see Section 48.4.1). This is because the log needs to be in the queue to allow the server to perform further actions on it - for example, to process a 'Log Request'.
3. The function **eCLD_ASCAddLog()** used to add a log to the local log queue (see Section 48.4.1) automatically sends a 'Log Notification' message to all bound Appliance Statistics cluster clients.

The log information from the 'Log Notification' message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION, which is generated when the message arrives from the server (Appliance Statistics events are described in Section 48.5). The log information from the message payload is contained in the above event - for details of this payload and the supplied log information, refer to Section 48.9.3.

Parent topic: Sending messages

Parent topic:*Appliance Statistics Cluster*

Log Operations on Server Appliance Statistics cluster functions are provided to allow the application on the cluster server (appliance) to perform the following local log operations:

- Add a log to the log queue
- Remove a log from the log queue
- Obtain a list of the logs in the log queue
- Obtain an individual log from the log queue

These operations are described in the sub-sections below.

Adding and Removing Logs A data log can be added to the local log queue (on the cluster server) using the function **eCLD_ASCAddLog()**. The log must be given an identifier and the UTC time at which the log was added must be specified. The length of the log, in bytes, must be less than the value of CLD_APPLIANCE_STATISTICS_ATTR_LOG_MAX_SIZE, which is defined in the **zcl_options.h** files (and must be less than or equal to 70).

The above function also sends a 'Log Notification' message to all bound Appliance Statistics cluster clients.

An existing log can be removed from the local log queue using the function **eCLD_ASCRemoveLog()**. The log is specified using its identifier.

Parent topic: Log Operations on Server

Obtaining Logs A list of the logs that are currently in the local log queue (on the cluster server) can be obtained by calling the function **eCLD_ASCGetLogsAvailable()**. This function provides the number of logs in the queue and a list of the log identifiers.

An individual log from the local log queue can be obtained using the function **eCLD_ASCGetLogEntry()**. The required log is specified by means of its identifier.

Normally, **eCLD_ASCGetLogsAvailable()**is called first to obtain a list of the available logs and then **eCLD_ASCGetLogEntry()**is called for each log.

Parent topic: Log Operations on Server

Parent topic:*Appliance Statistics Cluster*

Appliance statistics events The Appliance Statistics cluster has its own events that are handled through the callback mechanism described in *Chapter 3*. The cluster contains its own event handler. If a device uses this cluster then application-specific Appliance Statistics event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function will then be invoked when an Appliance Statistics event occurs and needs the attention of the application.

For an Appliance Statistics event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sCluster-CustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_ApplianceStatisticsCallBackMessage structure:

```
typedef struct
{
    uint8    u8CommandId;
    union
    {
        tsCLD_ASC_LogNotificationORLogResponsePayload
                                *psLogNotificationORLogResponsePayload;
        tsCLD_ASC_LogQueueResponseORStatisticsAvailablePayload
                                *psLogQueueResponseORStatisticsAvailabePayload;
        tsCLD_ASC_LogRequestPayload    *psLogRequestPayload;
    } uMessage;
} tsCLD_ApplianceStatisticsCallBackMessage;
```

When an Appliance Statistics event occurs, one of four command types could have been received. The relevant command type is specified through the u8CommandId field of the tsSM_CallBackMessage structure. The possible command types are detailed the tables below for events generated on a server and a client.

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_STATISTICS_	

CMD_LOG_REQUEST

|A 'Log Request' message has been received by the server (appliance)| |E_CLD_APPLIANCE_STATISTICS_

CMD_LOG_QUEUE_REQUEST

|A 'Log Queue Request' message has been received by the server (appliance)|

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_STATISTICS_	

CMD_LOG_NOTIFICATION

|A 'Log Notification' message has been received by the client| |E_CLD_APPLIANCE_STATISTICS_

CMD_LOG_RESPONSE

|A 'Log Response' message has been received by the client| |E_CLD_APPLIANCE_STATISTICS_

CMD_LOG_QUEUE_RESPONSE

|A 'Log Queue Response' message has been received by the client| |E_CLD_APPLIANCE_STATISTICS_

CMD_STATISTICS_AVAILABLE

|A 'Statistics Available' message has been received by the client|

Parent topic:*Appliance Statistics Cluster*

Functions The following Appliance Statistics cluster functions are provided:

1. eCLD_ApplianceStatisticsCreateApplianceStatistics
2. eCLD_ASCAddLog
3. eCLD_ASCRemoveLog
4. eCLD_ASCGetLogsAvailable
5. eCLD_ASCGetLogEntry
6. eCLD_ASCLogQueueRequestSend
7. eCLD_ASCLogRequestSend
8. eCLD_ASCLogQueueResponseORStatisticsAvailableSend
9. eCLD_ASCStatisticsAvailableSend
10. eCLD_ASCLogNotificationORLogResponseSend
11. eCLD_ASCLogNotificationSend

eCLD_ApplianceStatisticsCreateApplianceStatistics

```
teZCL_Status eCLD_ApplianceStatisticsCreateApplianceStatistics(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits,
        tsCLD_ApplianceStatisticsCustomDataStructure
*psCustomDataStructure);
```

Description This function creates an instance of the Appliance Statistics cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied tsZCL_ClusterInstance structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create an Appliance Statistics cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to *Appendix D.*

Note: This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in the *ZigBee Devices User Guide (JNUG3131).*

Note:

When used, this function must be the first Appliance Statistics cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length is automatically adjusted by the compiler using the following declaration:

```
uint8 au8ApplianceStatisticsAttributeControlBits
[(sizeof(asCLD_ApplianceStatisticsClusterAttributeDefinitions) / sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see see Section 6.1.2). In this case, this structure must contain the details of the Appliance Statistics cluster. This parameter can refer to a pre-filled structure called sCLD_ApplianceStatistics which is provided in the **ApplianceStatistics.h** file.
- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type tsCLD_ApplianceStatistics which defines the attributes of Appliance Statistics cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.
- *psCustomDataStructure* Pointer to a structure containing the storage for internal functions of the cluster (see Section 48.9.6).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_ASCAddLog

```
teZCL_CommandStatus eCLD_ASCAddLog(
        uint8 u8SourceEndPointId,
        uint32 u32LogId,
        uint8 u8LogLength,
        uint32 u32Time,
        uint8 *pu8LogData);
```

Description This function can be used on an Appliance Statistics cluster server to add a data log to the log queue. The function also sends out a 'Log Notification' message to all bound Appliance Statistics cluster clients.

The length of the data log, in bytes, must be less than the defined value of CLD_APPLIANCE_STATISTICS_ATTR_LOG_MAX_SIZE (which must be less than or equal to 70).

Parameter

- *u8SourceEndPointId* Number of the local endpoint on which the Appliance Statistics cluster server resides
- *u32LogId* Identifier of log
- *u8LogLength* Length of log, in bytes
- *u32Time* UTC time at which log was produced
- *pu8LogData* Pointer to log data

Returns

- E_ZCL_CMDS_SUCCESS
- E_ZCL_CMDS_FAIL
- E_ZCL_CMDS_INVALID_VALUE (log too long)
- E_ZCL_CMDS_INVALID_FIELD (NULL pointer to log data)
- E_ZCL_CMDS_INSUFFICIENT_SPACE

Parent topic: Functions

eCLD_ASCRemoveLog

```
teZCL_CommandStatus eCLD_ASCRemoveLog(
    uint8 u8SourceEndPointId,
    uint32 u32LogId);
```

Description This function can be used on an Appliance Statistics cluster server to remove the specified data log from the log queue.

Parameter

- *u8SourceEndPointId* Number of the local endpoint on which the Appliance Statistics cluster server resides
- *u32LogId* Identifier of log

Returns

- E_ZCL_CMDS_SUCCESS
- E_ZCL_CMDS_FAIL

Parent topic: Functions

eCLD_ASCGetLogsAvailable

```
teZCL_CommandStatus eCLD_ASCGetLogsAvailable(
        uint8 u8SourceEndPointId,
        uint32 *pu32LogId,
        uint8 *pu8LogIdCount);
```

Description This function can be used on an Appliance Statistics cluster server to obtain a list of the data logs in the log queue. The number of available logs and a list of their log IDs will be obtained.

Parameter

- *u8SourceEndPointId* Number of the local endpoint on which the Appliance Statistics cluster server resides
- *pu32LogId* Pointer to an area of memory to receive the list of 32-bit log IDs
- *pu8LogIdCount* Pointer to an area of memory to receive the number of logs in the queue

Returns

- E_ZCL_CMDS_SUCCESS
- E_ZCL_CMDS_FAIL

Parent topic: Functions

eCLD_ASCGetLogEntry

```
teZCL_CommandStatus eCLD_ASCGetLogEntry(
        uint8 u8SourceEndPointId,
        uint32 u32LogId,
        tsCLD_LogTable **ppsLogTable);
```

Description This function can be used on an Appliance Statistics cluster server to obtain the data log with the specified log ID.

Parameter

- *u8SourceEndPointId* Number of the local endpoint on which the Appliance Statistics cluster server resides
- *u32LogId* Log ID of the required data log
- *ppsLogTable* Pointer to a memory location to receive a pointer to the required data log

Returns

- E_ZCL_CMDS_SUCCESS
- E_ZCL_CMDS_FAIL
- E_ZCL_CMDS_NOT_FOUND (specified log not present)

Parent topic: Functions

eCLD_ASCLogQueueRequestSend

```
teZCL_Status eCLD_ASCLogQueueRequestSend(
uint8 u8SourceEndPointId,
uint8 u8DestinationEndPointId,
tsZCL_Address *psDestinationAddress,
uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on an Appliance Statistics cluster client to send a 'Log Queue Request' message to a cluster server (appliance), in order enquire about the availability of logs on the server.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_ASCLogRequestSend

```
teZCL_Status eCLD_ASCLogRequestSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ASC_LogRequestPayload *psPayload);
```

Description This function can be used on an Appliance Statistics cluster client to send a 'Log Request' message to a cluster server (appliance), in order request the data log with a specified log ID.

The function should normally be called after enquiring about log availability using the function **eCLD_ASCLogQueueRequestSend()** or after receiving an unsolicited 'Statistics Available' notification from the server.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload* Pointer to*a*structure containing the payload for the 'Log Request', including the relevant log ID (see Section 48.9.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_ASCLogQueueResponseORStatisticsAvailableSend

```
teZCL_Status eCLD_ASCLogQueueResponseORStatisticsAvailableSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        teCLD_ApplianceStatistics_ServerCommandId
eCommandId);
```

Description This function can be used on an Appliance Statistics cluster server to send a 'Log Queue Response' message (in reply to a 'Log Queue Request' message) or an unsolicited 'Statistics Available' message to a cluster client.

The command to be sent must be specified as one of:

- E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE
- E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values

- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *eCommandId* Enumeration indicating the command to be sent (see above and Section 48.8.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_ASCStatisticsAvailableSend

```
teZCL_Status eCLD_ASCStatisticsAvailableSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description This function can be used on an Appliance Statistics cluster server to send an unsolicited 'Statistics Available' message to a cluster client. The function is an alternative to **eCLD_ASCLogQueueResponseORStatisticsAvailableSend()**.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_ASCLogNotificationORLogResponseSend

```
teZCL_Status eCLD_ASCLogNotificationORLogResponseSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        teCLD_ApplianceStatistics_ServerCommandId
eCommandId,
        tsCLD_ASC_LogNotificationORLogResponsePayload
*psPayload);
```

Description This function can be used on an Appliance Statistics cluster server to send a 'Log Response' message (in reply to a 'Log Request' message) or an unsolicited 'Log Notification' message to a cluster client.

The command to be sent must be specified as one of:

- E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION
- E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *eCommandId* Enumeration indicating the command to be sent (see above and Section 48.8.3)
- *psPayload* Pointer to structure containing payload for message (see Section 48.9.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

eCLD_ASCLogNotificationSend

```
teZCL_Status eCLD_ASCLogNotificationSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ASC_LogNotificationORLogResponsePayload
*psPayload);
```

Description This function can be used on an Appliance Statistics cluster server to send an unsolicited 'Log Notification' message to a cluster client. The function is an alternative to **eCLD_ASCLogNotificationORLogResponseSend()**.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload* Pointer to structure containing payload for message (see Section 48.9.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

Parent topic: Functions

Parent topic:*Appliance Statistics Cluster*

Return codes The Appliance Statistics cluster functions use the ZCL return codes, listed in Section 7.2.

Parent topic:*Appliance Statistics Cluster*

Enumerations

'Attribute ID' enumerations The following structure contains the enumerations used to identify the attributes of the Appliance Statistics cluster.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_STATISTICS_ATTR_ID_LOG_MAX_SIZE = 0x0000,
    E_CLD_APPLIANCE_STATISTICS_ATTR_ID_LOG_QUEUE_MAX_SIZE
} teCLD_ApplianceStatistics_Cluster_AttrID;
```

Parent topic: Enumerations

'Client Command ID' enumerations The following enumerations are used in commands issued on a cluster client.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_REQUEST = 0x00,
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_REQUEST
} teCLD_ApplianceStatistics_ClientCommandId;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_REQUEST	'Log Request' message
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_REQUEST	'Log Queue Request' message

Parent topic: Enumerations

'Server Command ID' enumerations The following enumerations are used in commands issued on a cluster server.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION = 0x00,
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE,
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE,
    E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE
} teCLD_ApplianceStatistics_ServerCommandId;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION	A 'Log Notification' message
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE	A 'Log Response' message
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONS	A 'Log Queue Response' message
E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABL	A 'Statistics Available' message

Parent topic: Enumerations

Parent topic:*Appliance Statistics Cluster*

Structures

tsCLD_ApplianceStatisticsCallBackMessage For an Appliance Statistics event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsCLD_ApplianceStatisticsCallBackMessage structure:

```
typedef struct
{
    uint8    u8CommandId;
    union
    {
      tsCLD_ASC_LogNotificationORLogResponsePayload
                              *psLogNotificationORLogResponsePayload;
      tsCLD_ASC_LogQueueResponseORStatisticsAvailablePayload
                              *psLogQueueResponseORStatisticsAvailabePayload;
      tsCLD_ASC_LogRequestPayload *psLogRequestPayload;
    } uMessage;
} tsCLD_ApplianceStatisticsCallBackMessage;
```

where:

- u``8CommandId indicates the type of Appliance Statistics command that has been received, one of:
  - E_CLD_APPLIANCE_STATISTICS_CMD_LOG_REQUEST
  - E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_REQUEST
  - E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION
  - E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE
  - E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE
  - E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE
- u``Message is a union containing the command payload as one of (depending on the value of u8CommandId):
  - psLogNotificationORLogResponsePayload is a pointer to the payload of a 'Log Notification' or 'Log Response' message (see Section48.9.3)
  - psLogQueueResponseORStatisticsAvailabePayload is a pointer to the payload of a 'Log Queue Response' or 'Statistics Available' message (see Section 48.9.4)
  - psLogRequestPayload is a pointer to the payload of a 'Log Request' message (see Section 48.9.2)

Parent topic: Structures

tsCLD_ASC_LogRequestPayload This structure contains the payload for the 'Log Request' message.

```
typedef struct
{
    zuint32    u32LogId;
} tsCLD_ASC_LogRequestPayload;
```

where u32LogId is the identifier of the data log being requested.

Parent topic: Structures

tsCLD_ASC_LogNotificationORLogResponsePayload This structure contains the payload for the 'Log Notification' and 'Log Response' messages.

```
typedef struct
{
    zutctime     utctTime;
    zuint32      u32LogId;
    zuint32      u32LogLength;
    uint8       *pu8LogData;
} tsCLD_ASC_LogNotificationORLogResponsePayload;
```

where:

- utctTime is the UTC time at which the reported log was produced
- u32LogId is the identifier of the reported log
- u32LogLength is the length, in bytes, of the reported log
- pu8LogData is a pointer to an area of memory to receive the data of the reported log

Parent topic: Structures

tsCLD_ASC_LogQueueResponseORStatisticsAvailablePayload This structure contains the payload for the 'Log Queue Response' and 'Statistics Available' messages.

```
typedef struct
{
    zuint8      u8LogQueueSize;
    zuint32    *pu32LogId;
} tsCLD_ASC_LogQueueResponseORStatisticsAvailablePayload;
```

where:

- u8LogQueueSize indicates the number of logs currently in the log queue
- pu32LogId is a pointer to an area of memory to receive the sequence of 32-bit log IDs of the logs in the queue

Parent topic: Structures

tsCLD_LogTable This structure is used to store the details of a data log.

```
typedef struct
{
    zutctime     utctTime;
    uint32       u32LogID;
    uint8        u8LogLength;
    uint8       *pu8LogData;
} tsCLD_LogTable;
```

where:

- utctTime is the UTC time at which the log was produced
- u32LogId is the identifier of the log
- u32LogLength is the length, in bytes, of the log
- pu8LogData is a pointer to an area of memory to receive the data of the log

Parent topic: Structures

tsCLD_ApplianceStatisticsCustomDataStructure The Appliance Statistics cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress sReceiveEventAddress;
    tsZCL_CallBackEvent sCustomCallBackEvent;
    tsCLD_ApplianceStatisticsCallBackMessage sCallBackMessage;
#if (defined CLD_APPLIANCE_STATISTICS) && (defined APPLIANCE_STATISTICS_SERVER)
    tsCLD_LogTable asLogTable[CLD_APPLIANCE_STATISTICS_ATTR_LOG_QUEUE_MAX_SIZE];
#endif
} tsCLD_ApplianceStatisticsCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

Parent topic:*Appliance Statistics Cluster*

Compile-time options This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the Appliance Statistics cluster.

To enable the Appliance Statistics cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_APPLIANCE_STATISTICS
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define APPLIANCE_STATISTICS_SERVER
#define APPLIANCE_STATISTICS_CLIENT
```

The Appliance Statistics cluster contains macros that may be optionally specified at compile-time by adding some or all the following lines to the **zcl_options.h** file.

Global Attributes Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_APPLIANCE_STATISTICS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Maximum Log Size Add this line to configure the maximum size n, in bytes, of a data log:

```
#define CLD_APPLIANCE_STATISTICS_ATTR_LOG_MAX_SIZE n
```

The default value is 70 bytes, which is the upper limit on this value, and n must therefore not be greater than 70.

The same value must be defined on the cluster server and client.

Maximum Log Queue Length Add this line to configure the maximum number of logs n in a log queue:

```
#define CLD_APPLIANCE_STATISTICS_ATTR_LOG_QUEUE_MAX_SIZE n
```

The default value is 15, which is the upper limit on this value, and n must therefore not be greater than 15.

The same value must be defined on the cluster server and client.

Enable Insertion of UTC Time Add this line to enable the application to insert UTC time data into logs:

```
#define CLD_APPLIANCE_STATISTICS_ATTR_LOG_QUEUE_MAX_SIZE n
```

Disable APS Acknowledgements for Bound Transmissions Add this line to disable APS acknowledgements for bound transmissions from this cluster:

```
#define CLD_ASC_BOUND_TX_WITH_APS_ACK_DISABLED
```

Parent topic:*Appliance Statistics Cluster*

Part XII: Over-The-Air Upgrade This part comprises only one chapter:

- *Chapter 49* details the **OTA (Over-the-Air) Upgrade** cluster

OTA Upgrade cluster This chapter describes the Over-The-Air (OTA) Upgrade cluster. The OTA Upgrade cluster has a Cluster ID of 0x0019.

Note: This chapter assumes that the ZigBee 3.0 network consists of nodes which contain only one processor - such as a JN518x, K32W041, K32W061, MCXW71, or MCXW72 microcontroller. However, the OTA Upgrade cluster can also be used with dual-processor nodes (containing a JN518x, K32W041, K32W061, MCXW71, MCXW72 device and a coprocessor), as described in *Appendix F*.

Overview The Over-The-Air (OTA) Upgrade cluster provides the facility to upgrade (or downgrade or re-install) application software on the nodes of a ZigBee PRO network by:

1. Distributing the replacement software through the network (over the air) from a designated node.
2. Updating the software in a node with minimal interruption to the operation of the node.

The OTA Upgrade cluster acts as a server on the node that distributes the software and as a client on the nodes that receive software updates from the server. The cluster server receives the software from outside the network.

An application that uses the OTA Upgrade cluster must include the header files **zcl_options.h** and **OTA.h**.

The OTA Upgrade cluster is enabled by defining CLD_OTA in the **zcl_options.h** file. Further compile-time options for the OTA Upgrade cluster are detailed in Section 49.13.

When including the OTA Upgrade facility in your application, you should increase the CPU stack size from the default value (as described in Section 49.5).

Parent topic:*OTA Upgrade cluster*

OTA Upgrade Images in Internal Flash Memory This section provides guidance on how to organize Over-The-Air (OTA) upgrade images in internal Flash memory on the target node (OTA Upgrade client). The OTA Upgrade cluster is described in *Chapter 49*.

By default, OTA upgrade images are downloaded to a Flash memory device that is internal to the device of the OTA Upgrade client. However, the images can optionally be downloaded directly to devices internal Flash memory - this is enabled using the compile-time option, OTA_INTERNAL_STORAGE (see Section 49.13).

The function **eOTA_AllocateEndpointOTASpace()** is used in the application to allocate locations in Flash memory to store application images as part of the OTA upgrade process. The OTA code then uses these locations to store the upgrade image before switching to it, after validation.

There are two issues relating to OTA upgrade and Flash memory remapping:

- Whether the size of the OTA upgrade binary file is larger than the previous version, such that it must use another sector
- Where in the memory space the OTA image is written to

Consider the following cases.

We have a 154 KB image (5 sectors) and download a new image of the same size, starting at sector 8:

Case 1 (Sector 8)

Logical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Physical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Contents	Running image					Empty			New image					Empty		

When we switch to the new image, the physical sectors are moved in the memory map by the bootloader so that the new image becomes the running image and the previous running image becomes the old image. Only the sectors that must be moved are actually moved by the bootloader, and the other sectors are left alone:

Case 2

Logical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Physical Sector	8	9	10	11	12	5	6	7	0	1	2	3	4	13	14	15
Contents	New running image					Empty			Old image					Empty		

If we now download a further new image that is 161KB (6 sectors) in size, it would replace the old running image and be placed into logical sectors 8 to 13, which are physical sectors 0, 1, 2, 3, 4 and 13. However, this new image is then unusable because the physical sectors are not contiguous and the bootloader does not take this into account when it remaps the memory (if the new image was less than 160KB, there would be no problem).

The simple solution to this problem is to replace the remapping that the bootloader has chosen with our own remapping in which logical sector 13 becomes physical sector 5, thus allowing the new image to be stored in contiguous physical sectors (0 to 5).

Case 3

Logical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Physical Sector	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
Contents	Running image					Empty			New image						Empty	

However, this does not leave any space for permanent data and it also assumes that the new image is stored at logical sector 8.

You may choose to put the new image anywhere in the Flash memory (ZigBee allows this to be configured, and a user-developed solution is free to do what it requires). So you need to adjust the remapping to match. For example, if the OTA image was placed at logical sector 7:

Case 4

Logical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Physical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Contents	Running image					Empty		New image					Empty			

The purpose of this is to leave some sectors at the end of Flash memory for permanent data (otherwise you could always start the OTA image at sector 8).

In such cases, the sensible approach is to:

1. Calculate how much permanent data space is required and reserve the end sectors for this data.
2. Divide the remaining space into two equal blocks of sectors
3. Configure the OTA upgrade to start at the beginning of the second block of sectors.
4. Force the remapping to swap the two blocks, regardless of the actual image size.

Consider the example in which a user wants 64KB for permanent data, which requires 2 sectors. This leaves 14 sectors for applications, so we have two blocks of 7 sectors for each application (even though the application may be smaller than this):

Case 5

Logical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Physical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Contents	Running image					Empty		New image					Empty		Data	

To avoid any problem with the new image growing and needing 6 sectors rather than 5 sectors, we force the remapping to swap all 7 sectors over:

Case 6

Logical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Physical Sector	7	8	9	10	11	12	13	0	1	2	3	4	5	6	14	15
Contents	New running image					Empty		Old image					Empty		Data	

This leaves sectors 14 and 15 in a fixed location.

The code to achieve this is as follows:

```
`    if (u8CurrentImageSector > 0)`
`    { `
`      /* Remapping will not affect the current running image,`
`         which was already running in a continuous block at the base `
`         application Flash address */ `
`      vREG_SysWrite(REG_SYS_FLASH_REMAP,  0x0dcba987); `
`      vREG_SysWrite(REG_SYS_FLASH_REMAP2, 0xfe654321);`
`    }`
```

Parent topic:*OTA Upgrade cluster*

OTA Upgrade Cluster structure and attributes The attributes of the OTA Upgrade cluster are contained in the following structure, which is located only on cluster clients:

```
typedef struct
{
#ifdef OTA_CLIENT
    uint64 u64UgradeServerID;
    uint32 u32FileOffset;
    uint32 u32CurrentFileVersion;
    uint16 u16CurrentStackVersion;
    uint32 u32DownloadedFileVersion;
    uint16 u16DownloadedStackVersion;
    uint8  u8ImageUpgradeStatus;
    uint16 u16ManfId;
    uint16 u16ImageType;
    uint16 u16MinBlockRequestDelay;
#endif
    uint16 u16ClusterRevision;
} tsCLD_AS_Ota;
```

where:

- u64UgradeServerID contains the 64-bit IEEE/MAC address of the OTA Upgrade server for the client. This address can be fixed during manufacture or discovered during network formation/operation. If not pre-set, the default value is 0xFFFFFFFFFFFFFFFF. This attribute is mandatory.
- u32FileOffset contains the start address in local Flash memory of the upgrade image (that may be currently in transfer from server to client). This attribute is optional.
- u32CurrentFileVersion contains the file version of the firmware currently running on the client. This attribute is optional.
- u16CurrentStackVersion contains the version of the ZigBee stack currently running on the client. This attribute is optional.
- u32DownloadedFileVersion contains the file version of the downloaded upgrade image on the client. This attribute is optional.

- u16DownloadedStackVersion contains the version of the ZigBee stack for which the downloaded upgrade image was built. This attribute is optional.
- u8ImageUpgradeStatus contains the status of the client device in relation to image downloads and upgrades. This attribute is mandatory and the possible values are shown in the table below.

u8ImageUp	Status	Notes
0x00	Normal	Has not participated in a download/upgrade or the previous download/upgrade was unsuccessful
0x01	Download in progress	Client is requesting and successfully receiving blocks of image data from server
0x02	Download complete	All image data received and image saved to memory
0x03	Waiting to upgrade	Waiting for instruction from server to upgrade from the saved image
0x04	Count down	Server instructs the Client to count down to start of upgrade
0x05	Wait for more	Client is waiting for further upgrade image(s) from server - relevant to multi-processor devices, where each processor requires a different image
0x06 - 0xFF	Reserved	-

- u16ManfId contains the device's manufacturer code, assigned by the ZigBee Alliance. This attribute is optional.
- u16ImageType contains an image type identifier for the upgrade image that is currently being downloaded to the client or waiting on the client for the upgrade process to begin. When neither of these cases apply, the attribute is set to 0xFFFF. This attribute is optional.
- u16MinBlockRequestDelay is the minimum time, in seconds, that the local client must wait between submitting consecutive block requests to the server during an image download. It is used by the 'rate limiting' feature to control the average download rate to the client. The attribute can take values in the range 0 to OTA_BLOCK_REQUEST_DELAY_MAX_VALUE seconds, where this upper limit can be defined in the **zcl_options.h** file (see Section 49.13) - if undefined, its default value is 5 seconds. The value 0x0000 (default) indicates that the download can be performed at the full rate with no minimum delay between block requests. This attribute is optional.
- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

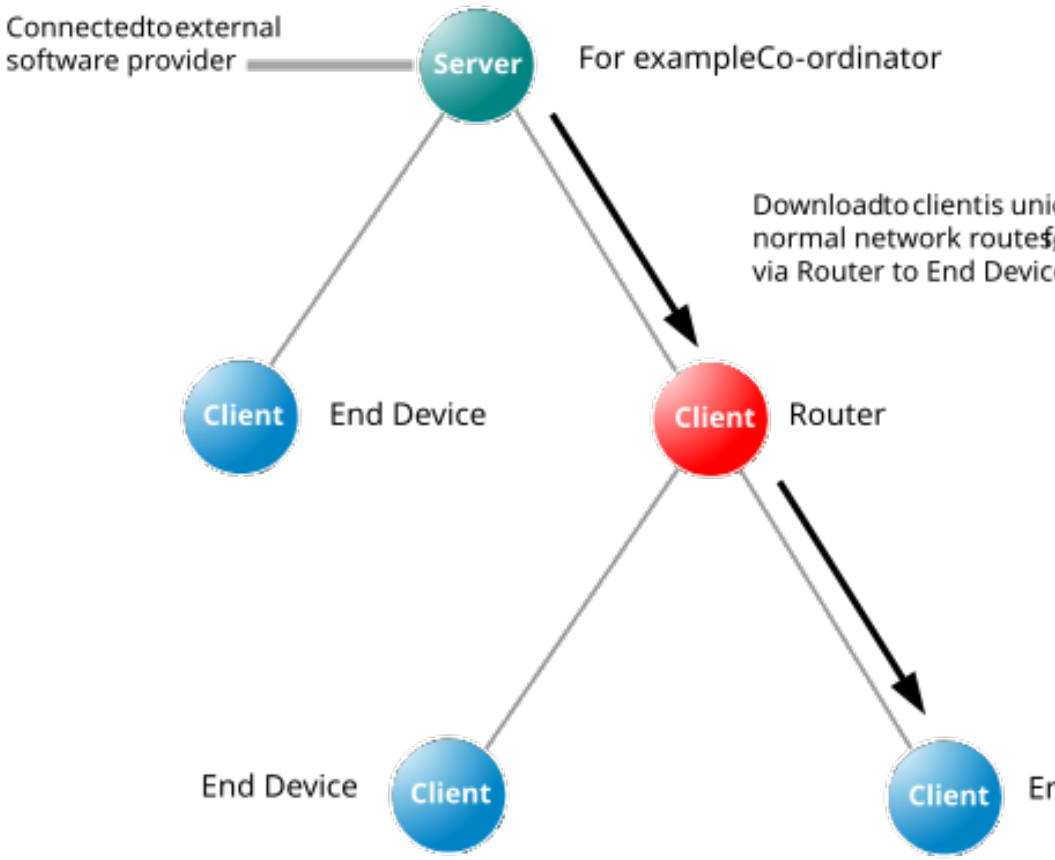
Thus, the OTA Upgrade cluster structure contains only two mandatory elements, u64UgradeServerID and u8ImageUpgradeStatus. The remaining elements are optional, each being enabled/disabled through a corresponding macro defined in the **zcl_options.h** file (see Section 49.13).

Parent topic:*OTA Upgrade cluster*

Basic Principles Over-the-Air (OTA) Upgrade allows the application software on a ZigBee node to be upgraded with minimal disruption to node operation and without physical intervention by the user/installer. For example, there is no need for a cabled connection to the node. Using this technique, the replacement software is distributed to nodes through the wireless network, allowing application upgrades to be performed remotely.

The software upgrade is performed from a node which acts as an OTA Upgrade cluster server, which is able to obtain the upgrade software from an external source. The nodes that receive the upgrade software act as OTA Upgrade cluster clients. The server node and client node(s) may be from different manufacturers.

The download of an application image from the server to the network is done on a per client basis and follows normal network routes (including routing via Routers). This is illustrated in the figure below.



OTA Download Example |

The upgrade application is downloaded into Flash memory internal to the device on the client node. Note that the upper section of Flash memory should normally be reserved for persistent data storage - for example, in an 8-sector Flash device, Sector 7 is used for persistent data storage, leaving Sectors 0-6 available to store application software.

The requirements of the devices which act as the OTA Upgrade cluster server and clients are detailed in the sub-sections below. See Connectivity Framework Reference Manual for details of the Non-Volatile Memory Manager (NVM).

OTA Upgrade Cluster Server The OTA Upgrade cluster server is a network node that distributes application upgrades to other nodes of the network (as well as performing its own functions). The server must, therefore, be connected to the provider of the upgrade software. The server would also usually be the Coordinator of the ZigBee network.

The server may need to store different upgrade images for different nodes (possibly from different manufacturers) and must have ample Flash memory space for this purpose. Therefore, the server must keep a record of the software required by each client in the network and the software version number that the client is currently on. When a new version of an application image becomes available, the server may notify the relevant client(s) or respond to poll requests for software upgrades from the clients (see Section 49.4.2 below).

Parent topic: Basic Principles

OTA Upgrade Cluster Client An OTA Upgrade cluster client is a node which receives software upgrades from the server and can be any type of node in a ZigBee network. However, an End Device client which sleeps is not always available to receive notifications of software upgrades from the server and must therefore, periodically poll the server for upgrades. In fact, all types of client can poll the server, if preferred.

During a software download from server to client, the upgrade image is transferred over the air in a series of data blocks. It is the responsibility of the client (and not the server) to keep track of the blocks received and then to validate the final image. The upgrade image is initially saved to the relevant sectors of Flash memory on the client. There must be enough Flash memory space on the client to store the upgrade image and the image of the currently running software.

An OTA upgrade image is downloaded into a Flash memory of the device, utilizing Flash sectors is currently not used for the running image.

Parent topic: Basic Principles

Parent topic:*OTA Upgrade cluster*

Application Requirements In order to implement OTA upgrades, the application images for the server and clients must be designed and built according to certain requirements.

These requirements include the following:

- Inclusion of the header files **zcl_options.h** and **OTA.h**
- Inclusion of the relevant #defines in the file **zcl_options.h**, as described in Section 49.13
- Specific application initialization requirements, as outlined in Section 49.6
- Use of the Non-Volatile Memory Manager (NVM) to preserve context data, as outlined in Section 49.8.5
- Organization of Flash memory, as outlined in Section 49.8.6
- It is necessary to remove references to the Certicom security certificate, as indicated in Section 49.13

Note: Some of above requirements differ between the server image, the first client image and client upgrade images. These differences are pointed out, where relevant, in Section 49.6 and Section 49.8.

In addition, you should increase the CPU stack size from the default value. With OTA Upgrade, the recommended stack size is 6000 bytes. This can be done by including the following line in your application makefile:

```
__stack_size = 6000;
```

Parent topic:*OTA Upgrade cluster*

Initialization Initialization of the various software components used with the OTA Upgrade cluster (see Section 49.5) must be performed in a particular order in the application code. The initialization could be incorporated in a function **APP_vInitialise()**, as is the case in the NXP *ZigBee PRO Application Template (JN-AN-1248)*.

Initialization must be performed in the following order:

1. The NVM module must first be initialized using the function **NvModuleInit()**.
2. The persistent data record(s) should then be initialized using the function and registered **NVM_RegisterDataSet()**.

3. The ZigBee PRO stack must now be started by first calling the function **ZPS_vSetOverrideLocalMacAddress()** to over-ride the existing MAC address, followed by **ZPS_eAplAfInit()** to initialize the Application Framework and then **ZPS_eAplZdoStartStack()** to start the stack.
4. The ZCL initialization function, **eZCL_Initialise()**, can now be called. An OTA Upgrade cluster instance should then be created using **eOTA_Create()**, followed by a call to **eOTA_UpdateClientAttributes()** or **eOTA_RestoreClientData()** on a client to initialize the cluster attributes.
5. The Flash programming of the OTA Upgrade cluster must now be initialized using the function **vOTA_FlashInit()**.
6. The required device endpoint(s) can now be registered (for example, a Simple Sensor device).
7. The function **eOTA_AllocateEndpointOTASpace()** must be called to allocate Flash memory space to an endpoint. The information provided to this function includes the numbers of the start sectors for storage of application images and the maximum number of sectors per image.
8. On the server, a set of client devices can be defined for which OTA upgrades are authorized - that is, a list of clients that are allowed to use the server for OTA upgrades. This client list is set up using the function **eOTA_SetServerAuthorisation()**.
9. For a client, a server must be found (provided this is a first-time start or a reboot with no persisted data, and so there is no record of a previous server address). This can be done by sending out a Match Descriptor Request using the function **ZPS_eAplZdpMatchDescRequest()**, described in the *ZigBee 3.0 Stack User Guide (JNUG3130)*. Once a server has been found, its address must be registered with the OTA Upgrade cluster using the function **eOTA_SetServerAddress()**.

The coding that is then required to implement OTA upgrade in the server and client applications is outlined in Section 49.7.

Parent topic:*OTA Upgrade cluster*

Implementing OTA Upgrade Mechanism The OTA upgrade mechanism is implemented in code as described below.

Note: The stack automatically handles part of an OTA upgrade and calls some of the OTA functions. However, if preferred, the application can handle all aspects of an OTA upgrade and filter all OTA data indications. In this case, the application must call all the relevant OTA functions (these are indicated below).

1. On the server, when a new client image is available for download, the function **eOTA_NewImageLoaded()** should be called to request the OTA Upgrade cluster to validate the image.

- Then, optionally, the function **eOTA_SetServerParams()** can be called to set the server parameter values for the new image. Otherwise, the default parameter values will be used.

2. The server must then notify the relevant client(s) of the availability of the new image. The notification method depends on the ZigBee node type of the client:

- **Coordinator or Router client:** The server can notify the Coordinator or a Router client directly by sending an Image Notify message to the client through a call to the function **eOTA_ServerImageNotify()**. This message can be unicast, multicast or broadcast. On arrival at a client, this message will trigger an Image Notify event. If the new software is required, the client can request the upgrade image by sending a Query Next Image Request to the server through a call to **eOTA_ClientQueryNextImageRequest()**.
- **All clients:** The server cannot notify an End Device client directly, since the End Device may be asleep when a notification message is sent. Therefore, an End Device client must

poll the server periodically (during wake periods) in order to establish whether new software is available. In fact, any client can implement polling of the server. The client does this by sending a Query Next Image Request to the server through a call to the function **eOTA_ClientQueryNextImageRequest()**.

  - On arrival at the server, the Query Next Image Request message triggers a Query Next Image Request event.

3. The server automatically replies to the request with a Query Next Image Response (the application can also send this response by calling the function **eOTA_ServerQueryNextImageResponse()**). The contents of this response message depend on whether the client is using notifications or polling:

- **Coordinator or Router client (notifications):** The response contains details of the upgrade image, such as manufacturer, image type, image size, and file version.
- **All clients (polling):** If upgrade software is available, the response reports success and the message contains details of the upgrade image, as indicated above. If no upgrade software is available, the response simply reports failure (the client must then poll again later).
  - On arrival at the client, the Query Next Image Response message triggers a Query Next Image Response event.

4. The OTA Upgrade cluster on the client now automatically requests the upgrade image one block at a time by sending an Image Block Request to the server (this request can also be sent by the application through a call to the function **eOTA_ClientImageBlockRequest()**). The maximum size of a block and the time interval between requests can both be configured in the header file **zcl-_options.h** - see Section 49.13.

- On arrival at the server, the Image Block Request message triggers an Image Block Request event.

5. The server automatically responds to each block request with an Image Block Response containing a block of data (the application can also send this response by calling the function **eOTA_ServerImageBlockResponse()**).

- On arrival at the client, the Image Block Response message triggers an Image Block Response event.

6. The client determines when the entire image has been received (by referring to the image size that was quoted in the Query Next Image Response before the download started). Once the final block of image data has been received, the client application should transmit an Upgrade End Request to the server (that is, by calling **eOTA_HandleImageVerification()**).

- This Upgrade End Request may report success or an invalid image. In the case of an invalid image, the image will be discarded by the client, which may initiate a new download of the image by sending a Query Next Image Request to the server.
- On arrival at the server, the Upgrade End Request message triggers an Upgrade End Request event.

Note: An Upgrade End Request may also be sent to the server during a download in order to abort the download.

7. The server replies to the request with an Upgrade End Response containing an instruction of when the client should use the downloaded image to upgrade the running software on the node (the message contains both the current time and the upgrade time, and hence an implied delay).

- On arrival at the client, the Upgrade End Response message triggers an Upgrade End Response event.

8. The client will then count down to the upgrade time (in the Upgrade End Response) and on reaching it, start the upgrade. If the upgrade time has been set to an indefinite value (represented by 0xFFFFFFFF), the client should poll the server for an Upgrade Command at least once per minute and start the upgrade once this command has been received.

9. Once triggered on the client, the upgrade process proceeds as follows (although the details will be manufacturer-specific):

a) A reboot of the device is initiated causing the default bootloader to run.

b) The bootloader scans through Flash looking for various image headers, including the running one and the one received via OTA. If the OTA image in newer, indicated by version number in the header, this is selected as the new running image.

Note: The client automatically invalidates the existing image and validates the new upgrade image once the allotted upgrade time is reached.

c) The new software image is then executed.

Query Jitter The 'query jitter' mechanism can be used to prevent a flood of replies to an Image Notify broadcast or multicast (Step 2 above). The server includes a number, n, in the range 1-100 in the notification. If interested in the image, the receiving client generates a random number in the range 1-100. If this number is greater than n, the client discards the notification, otherwise it responds with a Query Next Image Request. This results in only a fraction of interested clients responding to each broadcast/multicast and therefore helps to avoid traffic congestion.

Parent topic:*OTA Upgrade cluster*

Ancillary Features and Resources for OTA Upgrade As indicated in Section 49.5, in order to implement OTA upgrades, a number of other software features and resources are available. These are described in the sub-sections below.

Rate Limiting During busy periods when the OTA Upgrade server is downloading images to multiple clients, it is possible to prevent OTA traffic congestion by limiting the download rates to individual clients. This is achieved by introducing a minimum time-delay between consecutive Image Block Requests from a client - for example, if this delay is set to 500 ms for a particular client then after sending one block request to the server, the client must wait at least 500 ms before sending the next block request. This has the effect of restricting the average OTA download rate from the server to the client.

This 'block request delay' can be set to different values for different clients. This allows OTA downloads to be prioritized by granting more download bandwidth to some clients than to others. This delay for an individual client can also be modified by the server during a download, allowing the server to react in real-time to varying OTA traffic levels.

The implementation of the above rate limiting is described below and is illustrated in the Figure "Example of Rate limiting exchange" below.

'Block Request Delay' Attribute The download rate to an individual client is controlled using the optional attribute `u16MinBlockRequestDelay` of the OTA Upgrade cluster (see Section 49.3) on the client. This attribute contains the 'block request delay' for the client (described above), in milliseconds, and must be enabled on the client only (see below).

Note: The `u16MinBlockRequestDelay` attribute is the minimum time-interval between block requests. The application on the client can implement longer intervals between these requests (a slower download rate), if required.

Enabling the Rate Limiting Feature In order to use the rate limiting feature during an OTA upgrade, the macro OTA_CLD_ATTR_REQUEST_DELAY must be defined in the **zcl_options.h** file for both the participating client(s). This enables the `u16MinBlockRequestDelay` attribute in the OTA Upgrade cluster structure.

Implementation in the Server Application The application on the OTA Upgrade server device can control the OTA download rate to an individual client by remotely setting the value of the 'block request delay' attribute on the client. However, first the server must determine whether the client supports the rate limiting feature. The server can do this in either of two ways:

- It can attempt to read the u16MinBlockRequestDelay attribute in the OTA Upgrade cluster on the client - if rate limiting is not enabled on the client, this read will yield an error.
- It can check whether the first Image Block Request received from the client contains a 'block request delay' field - if present, this value is passed to the application in the event E_CLD_OTA_COMMAND_BLOCK_REQUEST.

The server can change the value of the 'block request delay' attribute on the client at any time, even during a download. To do this, the server includes the new attribute value in an Image Block Response with status OTA_STATUS_WAIT_FOR_DATA. This is achieved in the application code through a call to the function **eOTA_SetWaitForDataParams()** following an Image Block Request (indicated by an E_CLD_OTA_COMMAND_BLOCK_REQUEST event). The new attribute value specified in this function call is included in the subsequent Image Block Response and is automatically written to the OTA Upgrade cluster on the client.

The server may update the 'block request delay' attribute on a client multiple times during a download in order to react to changing OTA traffic conditions. If the server is downloading an image to only one client then it may choose to allow this download to proceed at the full rate (specified by a zero value of the attribute on the client). However, if two or more clients request downloads at the same time, the server may choose to limit their download rates (by setting the attribute to non-zero values on the clients). The download to one client can be given higher priority than other downloads by setting the attribute on this client to a lower value.

Implementation in the Client Application The application on the OTA Upgrade client device must control a millisecond timer (a timer with a resolution of one millisecond) to support rate limiting. This timer is used to time the delay between receiving an Image Block Response and submitting the next Image Block Request.

During an image download, a received Image Block Response with the status OTA_STATUS_WAIT_FOR_DATA may contain a new value for the 'block request delay' attribute (this type of response may arrive at the start of a download or at any time during the download). The client will automatically write this new value to the u16MinBlockRequestDelay attribute in the local OTA Upgrade cluster structure and will also generate the event E_ZCL_CBET_ENABLE_MS_TIMER (provided that the new attribute value is non-zero).

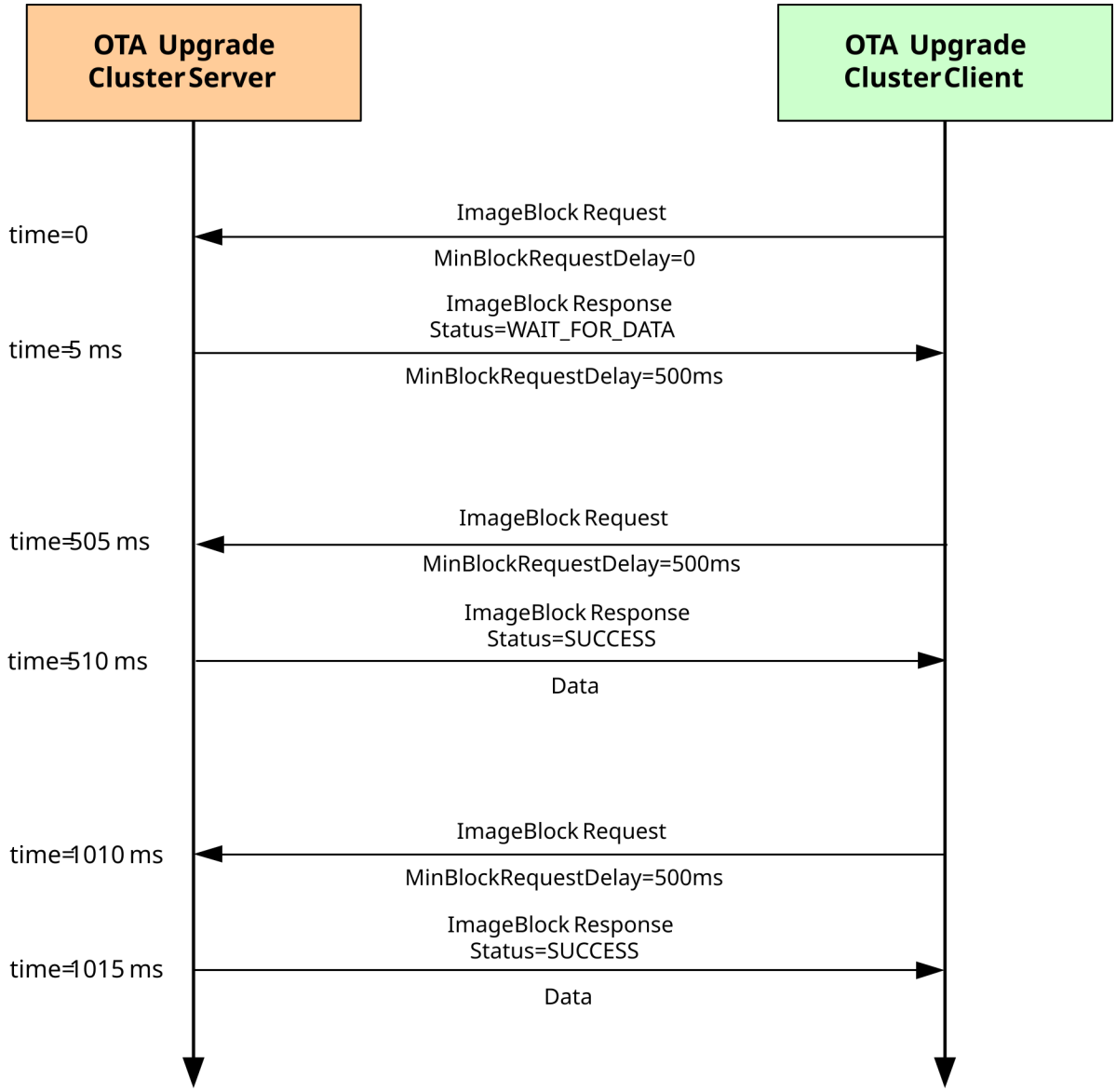
The E_ZCL_CBET_ENABLE_MS_TIMER event prompts the application to start the millisecond timer for a timed interval greater than or equal to the new value of the 'block request delay' attribute. The application can obtain this new attribute value (in milliseconds) from the event via:

```
sZCL_CallBackEvent.uMessage.u32TimerPeriodMs
```

The millisecond timer is started for a particular timed interval. The expiry of this timer is indicated by an E_ZCL_CBET_TIMER_MS event, which is handled as described in Section 3.2. The client will then send the next Image Block Request.

After sending an Image Block Request:

- If the client now generates an E_ZCL_CBET_DISABLE_MS_TIMER event, this indicates that the last of the Image Block Request (for the required image) has been sent and the application should disable the millisecond timer.
- Otherwise, the application must start the next timed interval (until the next request).
  Example of Rate limiting exchange



Parent topic: Ancillary Features and Resources for OTA Upgrade

Device-Specific File Downloads An OTA Ugrade client can request a file (from the server) that is specific to the client device. This file may contain non-firmware data such as security credentials, configuration data or log data. The process of making this request and receiving the file is described in the table below for both the client and server sides.

	On Client	On Server
1	Client application sends a Query Specific File Request to the server through a call to **eOTA_ClientQuerySpecificFileRequest()**.	
2		On arrival at the server, the Query Specific File Request triggers the event E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_REQUEST.
3		Server automatically replies to the request with a Query Specific File Response - the application can also send a response using **eOTA_ServerQuerySpecificFileResponse()**.
4	On arrival at the client, the Query Specific File Response triggers the event E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_	
5	Client obtains status from Query Specific File Response. If status is SUCCESS, the client automatically requests the device-specific file one block at a time by sending Image Block Requests to the server.	
6		On arrival at the server, each Image Block Request triggers an Image Block Request event.
7		Server automatically responds to each block request with an Image Block Response containing a block of device-specific file data.
8	After receiving each Image Block Response, the client generates the event E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_F	
9	A callback function is invoked on the client to handle the event and store the data block (it is the responsibility of the application to store the data in a convenient place).	
10	Client determines when the entire file has been received (by referring to the file size that was quoted in the Query Specific File Response before the download started). Once all the file blocks have been received:	

- E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_COMPLETE event is generated by the client to indicate that the file transfer is complete.
- The file can optionally be verified by application.
- Client sends an Upgrade End Request to the server to indicate that the download is complete, where this request is the result of an application call to the function **eOTA_SpecificFileUpgradeEndRequest()**.

|| |**11**||On arrival at the server, the Upgrade End Request triggers an Upgrade End Request event.| |**12**||Server may reply to the Upgrade End Request with an Upgrade End Response containing an instruction of when the client should use the device-specific file (the message contains both the current time and the upgrade time, and hence an implied delay) - see Footnotes 1 and 2 below.| |**13**|On arrival at the client, the Upgrade End Response triggers an Upgrade End Response event - see Footnotes 1 and 2 below.|| |**14**|Client will then count down to the upgrade time (in the Upgrade End Response) and, on reaching it, will generate the event

E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_USE_NEW_FILE. Finally, it is the responsibility of the application to use device-specific file as appropriate.| |

Footnotes

1. For a device-specific file download, it is not mandatory for the server to send an Upgrade End Response to the client. In the case of a client which has just finished retrieving a log file from the server, the Upgrade End Response may not be needed. However, if the client has just retrieved a file containing security credentials or configuration data, the Upgrade End Response may be needed to notify the client of when to apply the file. The decision of whether to send an Upgrade End Response for a device-specific file download is manufacturer-specific.

2. If an Upgrade End Response is not received from the server, the client will perform 3 retries to get the response. If it still does not receive a response, the client will generate the event E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_NO_UPGRADE_END_RESPONSE.

Parent topic: Ancillary Features and Resources for OTA Upgrade

Image Block Size and Fragmentation An OTA Upgrade image is normally requested by the OTA Upgrade client one block at a time. The ZigBee frame for the OTA transfer contains various header data as well as payload data and, for this reason, the payload data is limited to about 48 bytes. Therefore, to transfer one image block per frame, the block size must be restricted to 48 bytes or less. The maximum block size can be configured at compile-time through the OTA_MAX_BLOCK_SIZE define in the **zcl_options.h** file (see Section 49.13).

A block size of greater than 48 bytes can be used but the image block will need to be transferred across two or more ZigBee frames. In this case, fragmentation must be enabled in which the image block data that is assembled in an APDU (Application Protocol Data Unit) on the server is fragmented into multiple NPDUs (Network Protocol Data Unit) for OTA transfer, where one NPDU is transferred in a single ZigBee frame. Fragmentation is enabled on the OTA Upgrade server and client using network parameters of the ZigBee PRO stack, as follows:

- **On the server:** Set the parameter *Maximum Number of Transmitted Simultaneous Fragmented Messages* to a non-zero value to allow transmitted messages to be fragmented.
- **On the client:** Set the parameter *Maximum Number of Received Simultaneous Fragmented Messages* to a non-zero value to allow received fragmented messages to be re-assembled.

The network parameter values are set using the ZPS Configuration Editor and are described in the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

Note: **Note:**The 48-byte limit on the payload data in a ZigBee frame is also applicable when image data is requested and transferred one page at a time (see Section 49.8.4). In this case, fragmentation may need to be enabled.

The maximum APDU size must always be greater than the size of an Image Block Response. It is set through the APDU *Size* parameter of the PDU Manager, where this parameter is amongst the Advanced Device Parameters that can be configured using the ZPS Configuration Editor.

Depending on the image block size, fragmentation is not always an efficient way of transferring image blocks, as the payload of the final NPDU fragment may contain little data and be mostly empty. For example, if the image block size is set to 64 bytes and fragmentation is enabled, each block is transferred in two ZigBee frames, the first may contain 48 bytes of data and the second may contain only 16 bytes of data, leaving 32 empty bytes in the payload. In contrast, if the block size is set to 48 bytes without fragmentation, two consecutive frames would carry 96 bytes of data, and the image transfer would require fewer frames. This is particularly important when transferring an application image to a battery-powered End Device that needs to conserve energy.

Parent topic: Ancillary Features and Resources for OTA Upgrade

Page Requests An OTA Upgrade client normally requests image data from the server one block at a time, by sending an Image Block Request when it is ready for the next block. The number of requests can be reduced by requesting the image data one page at a time, where a page may contain many blocks of data. Requesting data by pages reduces the OTA traffic and, in the case of battery-powered client device, extends battery life.

A page of data is requested by sending an Image Page Request to the server. This request contains a page size, which indicates the number of data bytes that should be returned by the server following the request (and before the next request is sent, if any). The server still sends the data one block at a time in Image Block Responses. The Image Page Request also specifies the maximum number of bytes that the client device can receive in any one OTA message and the block size must therefore not exceed this limit (in general, the page size should be a multiple of this limit).

It is the responsibility of the client to keep track of the amount of data so far received since the last Image Page Request was issued - this count is updated after each Image Block Response received. Once this count reaches the page size in the request, the client will issue the next Image Page Request (if the download is not yet complete).

During a download that uses page requests:

- If the client fails to receive one or more of the requested blocks then the next Image Page Request will request data starting from the offset which corresponds to the first missing block.
- If the client fails to receive all the blocks requested in an Image Page Request then the same request will be repeated up to two more times - if the requested data still fails to arrive, the client will switch to using Image Block Requests to download the remaining image data.

An Image Page Request also contains a 'response spacing' value. This indicates the minimum time-interval, in milliseconds, that the server should insert between consecutive Image Block Responses. If the client is a sleepy End Device, it may specify a long response spacing so that it can sleep between consecutive Image Block Responses, or it may specify a short response spacing so that it can quickly receive all blocks requested in a page and sleep between consecutive Image Page Requests.

The implementation of the above page requests in an application is described below. The OTA image download process using page requests is similar to the one described in Section 49.7, except the client submits Image Page Requests to the server instead of Image Block Requests.

Enabling the Page Requests Feature In order to use page requests, the macro OTA_PAGE_REQUEST_SUPPORT must be defined in the **zcl_options.h** file for the server and client.

In addition, values for the page size and response spacing can also be defined in this file for the client (if non-default values are required) - see below and Section 49.13.

Implementation in the Server Application The application on the OTA Upgrade server device must control a millisecond timer (a timer with a resolution of one millisecond) to support page requests. This timer is used to implement the 'response spacing' specified in an Image Page Request - that is, to time the interval between the transmissions of consecutive Image Block Responses (sent out in response to the Image Page Request).

When the server receives an Image Page Request, it will generate the event E_ZCL_CBET_ENABLE_MS_TIMER to prompt the application to start the millisecond timer for a timed interval equal in value to the 'response spacing' in the request. The application can obtain this value (in milliseconds) from the event via:

```
sZCL_CallBackEvent.uMessage.u32TimerPeriodMs
```

The millisecond timer is started for a particular timed interval. The expiry of this timer is indicated by an E_ZCL_CBET_TIMER_MS event, which is handled as described in Section 3.2. The server will then send the next Image Block Response.

After sending an Image Block Response:

- If the server now generates an E_ZCL_CBET_DISABLE_MS_TIMER event, this indicates that the last of the Image Block Responses (for the Image Page Request) has been sent and the application should disable the millisecond timer.
- Otherwise, the application must start the next timed interval (until the next response).

Implementation in the Client Application There is nothing specific to do in the client application to implement page requests. Provided that page requests have been enabled in the **zcl_options.h** file for the client (see above), page requests will be automatically implemented by the stack instead of block requests for OTA image downloads. The page size (in bytes) and response spacing (in milliseconds) for these requests can be specified through the following macros in the **zcl_options.h** file (see Section 49.13):

- OTA_PAGE_REQ_PAGE_SIZE
- OTA_PAGE_REQ_RESPONSE_SPACING

The default values are 512 bytes and 300 ms, respectively.

However, the client application can itself submit an Image Page Request to the server by calling the function **eOTA_ClientImagePageRequest()**. In this case, the page size and response spacing are specified in the Image Page Request payload structure as part of this function call.

The client handles the resulting Image Block Responses as described in Section 49.7 for standard OTA downloads.

Parent topic: Ancillary Features and Resources for OTA Upgrade

Persistent Data Management The OTA Upgrade cluster on a client requires context data to be preserved in non-volatile memory to facilitate a recovery of the OTA Upgrade status following a device reboot. The Non-Volatile Memory Manager (NVM) module should be used to perform this data saving and recovery. The NVM module is implemented as described in the *Connectivity Framework Reference Manual*.

Persistent data is normally be stored in the upper sector of the devices Flash memory. Thus, when the NVM module is initialized, these sectors should be specified (just these sectors should be managed by the NVM module).

When it needs to save context data, the OTA Upgrade cluster will generate the event E_CLD_OTA_INTERNAL_COMMAND_SAVE_CONTEXT, which will also contain the data to be saved. A user-defined callback function can then be invoked to perform the data storage using functions of the NVM module.

The OTA Upgrade cluster is implemented for an individual application/endpoint. Therefore, the NVM module should also be implemented per endpoint. The following code illustrates the reservation of memory space for persistent data per endpoint.

```
typedef struct
{
        uint8 u8Endpoints[APP_NUM_OF_ENDPOINTS];
        uint8 eState; // Current application state to re-instate
  tsOTA_PersistedData sPersistedData[APP_NUM_OF_ENDPOINTS];
} tsDevice;
PUBLIC tsDevice s_sDevice;
```

If a client is restarted and persisted data is available on the device, the OTA Upgrade cluster data should be restored using the function **eOTA_RestoreClientData()**.

Parent topic: Ancillary Features and Resources for OTA Upgrade

Flash Memory Organization Flash memory should be organized such that the application images are stored from Sector 0 and, if required, persistent data is stored in the final sectors.

The storage of applications and persistent data in Flash memory is described further below. Guidance on the organization of OTA upgrade applications in the devices internal Flash memory is also provided Appendix.

Application Images As part of application initialization (see Section 49.6), the OTA Upgrade cluster must be informed of the storage arrangements for application images in Flash memory. This is done through the function **eOTA_AllocateEndpointOTASpace()**, which applies to a specified endpoint (normally the endpoint of the application which calls the function). The information provided via this function includes:

- Start sector for each image that can be stored (specified through an array with one element per image). s
- Number of images for the endpoint (the maximum number of images per endpoint is specified in the **zcl_options.h** file - see Section 49.13)
- Maximum number of sectors per image
- Type of node (server or client)
- Public key for signed images

Persistent Data The storage of persistent data is handled by the NVM module (see Section 49.8.5) and the sector used is specified as part of the NVM initialization through **NvModuleInit()** - the final sector of Flash memory should be specified.

Parent topic: Ancillary Features and Resources for OTA Upgrade

Low-Voltage Flag An OTA Upgrade cluster client should not attempt to participate in an OTA upgrade if the supply voltage to the host hardware device is low (below the normal operating voltage for the device). On the device, sufficient voltage is required to write to the internal Flash There may be a number of reasons for a sudden drop in supply voltage - for example, the voltage on a battery-powered node may fall when the battery is near the end of its life.

The OTA Upgrade cluster incorporates a mechanism which, if enabled, stops the cluster client from sending Image Block Requests to the server when the local supply voltage becomes low. This mechanism allows the application to set a low-voltage flag which, when set, automatically suspends the block requests. When the flag is cleared, the block requests are automatically resumed.

If required, use of the low-voltage flag and associated mechanism must be enabled at compile-time by including the following line in the **zcl_options.h** file:

```
#define OTA_UPGRADE_VOLTAGE_CHECK
```

It is the responsibility of the application to check the supply voltage. This check is system-specific and may be performed periodically or using a voltage monitoring feature - for example, on the device, the Supply Voltage Monitor (SVM) can be used, which is described in the MCUXpresso SDK API Reference Manual.

The application can use the function **vOTA_SetLowVoltageFlag()** to configure the low-voltage flag. This function is detailed in Section 49.10.3.

When a low voltage is detected, the application should make the following function call to set the low-voltage flag and suspend Image Block Requests:

```
vOTA_SetLowVoltageFlag(TRUE);
```

When the voltage is restored to a normal level, the application should make the following function call to clear the low-voltage flag and resume Image Block Requests:

```
vOTA_SetLowVoltageFlag(FALSE);
```

Parent topic: Ancillary Features and Resources for OTA Upgrade

Parent topic:*OTA Upgrade cluster*

OTA Upgrade events The events that can be generated on an OTA Upgrade cluster server or client are defined in the structure teOTA_UpgradeClusterEvents (see Section 49.12.2). The events are listed in the table below, which also indicates on which side of the cluster (server or client) the events can occur:

Cluster Side(s)	Event
Server	E_CLD_OTA_CO
E_CLD_OTA_COMMAND_BLOCK_REQUEST	
E_CLD_OTA_COMMAND_PAGE_REQUEST	
E_CLD_OTA_COMMAND_UPGRADE_END_REQUEST	
E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_REQUEST	
E_CLD_OTA_INTERNAL_COMMAND_SEND_UPGRADE_END_RESPONSE	
	E_CLD_OTA_IN
Client	E_CLD_OTA_CO
E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE	
E_CLD_OTA_COMMAND_BLOCK_RESPONSE	
E_CLD_OTA_COMMAND_UPGRADE_END_RESPONSE	
E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_RESPONSE	
E_CLD_OTA_INTERNAL_COMMAND_TIMER_EXPIRED	
E_CLD_OTA_INTERNAL_COMMAND_POLL_REQUIRED	
E_CLD_OTA_INTERNAL_COMMAND_RESET_TO_UPGRADE	
E_CLD_OTA_INTERNAL_COMMAND_SAVE_CONTEXT	
E_CLD_OTA_INTERNAL_COMMAND_OTA_DL_ABORTED	
E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_BLOCK_RESPONSE	
E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_DL_ABORT	
E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_IMAGE_DL_COMPLETE	
E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_SWITCH_TO_NEW_IMAGE	
E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_BLOCK_RESPONSE	
E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_COMPLETE	
E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_ABORT	
E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_USE_NEW_FILE	
E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_NO_UPGRADE_END_RESPONSE	
E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE_ERROR	
E_CLD_OTA_INTERNAL_COMMAND_VERIFY_SIGNER_ADDRESS	
E_CLD_OTA_INTERNAL_COMMAND_RCVD_DEFAULT_RESPONSE	
E_CLD_OTA_INTERNAL_COMMAND_VERIFY_IMAGE_VERSION	
E_CLD_OTA_INTERNAL_COMMAND_SWITCH_TO_UPGRADE_DOWNGRADE	
E_CLD_OTA_INTERNAL_COMMAND_REQUEST_QUERY_NEXT_IMAGES	

Table 6 – continued from previous page

Cluster Side(s)	Event
E_CLD_OTA_INTERNAL_COMMAND_OTA_START_IMAGE_VERIFICATION_IN_LOW_PRIORITY	
E_CLD_OTA_INTERNAL_COMMAND_FAILED_VALIDATING_UPGRADE_IMAGE	
	E_CLD_OTA_IN
E_CLD_OTA_BLOCK_RESPONSE_TAG_OTHER_THAN_UPGRADE_IMAGE	
Both	E_CLD_OTA_IN
E_CLD_OTA_INTERNAL_COMMAND_FREE_FLASH_MUTEX	

OTA Upgrade events are treated as ZCL events. Thus, an event is received by the application, which wraps the event in a tsZCL_CallBackEventstructure and passes it into the ZCL using the function **vZCL_EventHandler()** - for further details of ZCL event processing, refer to *Chapter 3*.

The above events are outlined in the sub-sections below.

Server-side Events

- **E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_REQUEST**
  - This event is generated on the server when a Query Next Image Request is received from a client to enquire whether a new application image is available for download. The event may result from a poll request from the client or may be a consequence of an Image Notify message previously sent by the server. The server reacts to this event by returning a Query Next Image Response.
- **E_CLD_OTA_COMMAND_BLOCK_REQUEST**
  - This event is generated on the server when an Image Block Request is received from a client to request a block of image data as part of a download. The application reacts to this event by returning an Image Block Response containing a data block.
- **E_CLD_OTA_COMMAND_PAGE_REQUEST**
  - This event is generated on the server when an Image Page Request is received from a client to request a page of image data as part of a download.
- **E_CLD_OTA_COMMAND_UPGRADE_END_REQUEST**
  - This event is generated on the server when an Upgrade End Request is received from a client to indicate that the complete image has been downloaded and verified. The application reacts to this event by returning an Upgrade End Response.
- **E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_REQUEST**
  - This event is generated on the server when a Query Specific File Request is received from a client to request a particular application image. The server reacts to this event by returning a Query Specific File Response.
- **E_CLD_OTA_INTERNAL_COMMAND_SEND_UPGRADE_END_RESPONSE**
  - This event is generated on the server to notify the application that the stack is going to send an Upgrade End Response to a client. No specific action is required by the application on the server.

Parent topic:OTA Upgrade events

Client-side Events

- **E_CLD_OTA_COMMAND_IMAGE_NOTIFY**
  - This event is generated on the client when an Image Notify message is received from the server to indicate that a new application image is available for download. If the client decides to download the image, the application should react to

this event by sending a Query Next Image Request to the server using the function **eOTA_ClientQueryNextImageRequest()**.

- **E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE**
  - This event is generated on the client when a Query Next Image Response is received from the server (in response to a Query Next Image Request) to indicate whether a new application image is available for download. If a suitable image is reported, the client initiates a download by sending an Image Block Request to the server.
- **E_CLD_OTA_COMMAND_BLOCK_RESPONSE**
  - This event is generated on the client when an Image Block Response is received from the server (in response to an Image Block Request) and contains a block of image data which is part of a download. Following this event, the client can request the next block of image data by sending an Image Block Request to the server or, if the entire image has been received and verified, the client can close the download by sending an Upgrade End Request to the server.
- **E_CLD_OTA_COMMAND_UPGRADE_END_RESPONSE**
  - This event is generated on the client when an Upgrade End Response is received from the server (in response to an Upgrade End Request) to confirm the end of a download. This event contains the time delay before the upgrade of the running application must be performed.
- **E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_RESPONSE**
  - This event is generated on the client when a Query Specific File Response is received from the server (in response to a Query Specific File Request) to indicate whether the requested application image is available for download.
- **E_CLD_OTA_INTERNAL_COMMAND_TIMER_EXPIRED**
  - This event is generated on the client when the local one-second timer has expired. It is an internal event and is not passed to the application.
- **E_CLD_OTA_INTERNAL_COMMAND_POLL_REQUIRED**
  - This event is generated on the client to prompt the application to poll the server for a new application image by calling the function **eOTA_ClientQueryNextImageRequest()**.
- **E_CLD_OTA_INTERNAL_COMMAND_RESET_TO_UPGRADE**
  - This event is generated on the client to notify the application that the stack is going to reset the device. No specific action is required by the application.
- **E_CLD_OTA_INTERNAL_COMMAND_SAVE_CONTEXT**
  - This event prompts the client application to store context data in Flash memory. The data to be stored is passed to the application within this event.
- **E_CLD_OTA_INTERNAL_COMMAND_OTA_DL_ABORTED**
  - This event is generated on a client if the received image is invalid or the client has aborted the image download. This allows the application to request the new image again.
- **E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_BLOCK_RESPONSE**
  - This event is generated on the client when an Image Block Response is received from the server in response to an Image Block Request for a device-specific file. The event contains a block of file data which is part of a download. Following this event, the client stores the data block in an appropriate location and can request the next block of file data by sending an Image Block Request to the server (if the complete image has not yet been received and verified).
- **E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_COMPLETE**

  - This event is generated on the client when the final Image Block Response of a device-specific file download has been received from the server - the event indicates that all the data blocks that make up the file have been received.

- **E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_USE_NEW_FILE**
  - This event is generated on the client following a device-specific file download to indicate that the file can now be used by the client. At the end of the download, the server sends an Upgrade End Response that may include an 'upgrade time' - this is the UTC time at which the new file can be applied. Thus, on receiving this response, the client starts a timer and, on reaching the upgrade time, generates this event.
- **E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_ABORT**
  - This event is generated to indicate that the OTA Upgrade cluster needs to abort a device-specific file download. Following this event, the application should discard data that has already been received as part of the aborted download.
- **E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_NO_UPGRADE_**

  END_RESPONSE
  - This event is generated when no Upgrade End Response has been received for a device-specific file download. The client makes three attempts to obtain an Upgrade End Response. If no response is received, the client raises this event.

Note: For a device-specific file download, it is not mandatory for the server to send an Upgrade End Response. The decision of whether to send the Upgrade End Response is manufacturer-specific.

- **E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE_ERROR**
  - This event is generated on the client when a Query Next Image Response message is received from the server, in response to a Query Next Image Request with a status of Invalid Image Size.
- **E_CLD_OTA_INTERNAL_COMMAND_RCVD_DEFAULT_RESPONSE**
  - This event is generated on the client when a default response message is received from the server, in response to a Query Next Image Request, Image Block Request or Upgrade End Request. This is an internal ZCL event that results in an OTA download being aborted, thus activating the callback function for the E_CLD_OTA_INTERNAL_COMMAND_OTA_DL_ABORTED event.
- **E_CLD_OTA_INTERNAL_COMMAND_VERIFY_IMAGE_VERSION**
  - This event is generated to prompt the application to verify the image version received in a Query Next Image Response. This event allows the application to verify that the new upgrade image has a valid image version. After checking the image version, the application should set the status field of the event to E_ZCL_SUCCESS (valid version) or E_ZCL_FAIL (invalid version).
- **E_CLD_OTA_INTERNAL_COMMAND_SWITCH_TO_UPGRADE_**

  DOWNGRADE
  - This event is generated to prompt the application to verify the image version received in an upgrade end response. This event allows the application to verify that the new upgrade image has a valid image version.
  - After checking the image version, the application should set the status field of the event to E_ZCL_SUCCESS (valid version) or E_ZCL_FAIL (invalid version).
- **E_CLD_OTA_INTERNAL_COMMAND_FAILED_VALIDATING_UPGRADE_**

  IMAGE
  - This event is generated on the client when the validation of a new upgrade image fails. This validation takes place when the upgrade time is reached.

- **E_CLD_OTA_INTERNAL_COMMAND_FAILED_COPYING_SERIALIZATION_DATA**
  - This event is generated on the client when the copying of serialisation data from the active image to the new upgrade image fails. This process takes place after image validation (if applicable) is completed successfully.
- **E_CLD_OTA_BLOCK_RESPONSE_TAG_OTHER_THAN_UPGRADE_**

  IMAGE
  - This event is generated on the client when an Image Block Response is received from the server but the response contains a block of data that is not upgrade image data (it may contain tags such as an integrity code or ECDA signature).Thus, this event can help the application to process tags or data other than upgrade image data.

Parent topic:OTA Upgrade events

Server-side and Client-side Events

- **E_CLD_OTA_INTERNAL_COMMAND_LOCK_FLASH_MUTEX**
  - This event prompts the application to lock the mutex used for accesses to external Flash memory (via the SPI bus).
- **E_CLD_OTA_INTERNAL_COMMAND_FREE_FLASH_MUTEX**
  - This event prompts the application to unlock the mutex used for accesses to external Flash memory (via the SPI bus).

Parent topic:OTA Upgrade events

Parent topic:*OTA Upgrade cluster*

Functions The OTA Upgrade cluster functions that are provided in the NXP implementation of the ZCL are divided into the following three categories:

- General functions (used on server and client) - see Section 49.10.1
- Server functions - see Section 49.10.2
- Client functions - see Section 49.10.3

General Functions The following OTA Upgrade cluster functions can be used on the cluster server and the cluster client:

1. eOTA_Create
2. vOTA_FlashInit
3. eOTA_AllocateEndpointOTASpace
4. vOTA_GenerateHash
5. eOTA_GetCurrentOtaHeader

eOTA_Create

```
teZCL_Status eOTA_Create(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 u8Endpoint,
    uint8 *pu8AttributeControlBits,
```

(continues on next page)

(continued from previous page)

```
    tsOTA_Common *psCustomDataStruct);
```

Description This function creates an instance of the OTA Upgrade cluster on the specified endpoint. The cluster instance can act as a server or a client, as specified. The shared structure of the device associated with cluster must also be specified.

The function must be the first OTA function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16)
- *bIsServer*: Side of cluster to be implemented on this device:
  - TRUE - Server
  - FALSE - Client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster (see Section 6.1.2) - this structure must contain the details of the OTA Upgrade cluster
- *pvEndPointSharedStructPtr*: Pointer to shared device structure for relevant endpoint (depends on device type, e.g. Door Lock)
- *u8Endpoint*: Number of endpoint with which cluster will be associated
- *pu8AttributeControlBits*: Pointer to an array of bitmaps, one for each attribute in the relevant cluster - for internal cluster definition use only, array should be initialised to 0
- *tpsCustomDataStruct*: Pointer to structure containing custom data for OTA Upgrade cluster (see Section 49.11.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: General Functions

vOTA_FlashInit

```
void vOTA_FlashInit(void *pvFlashTable,
    tsNvmDefs *psNvmStruct);
```

Description This function initializes the Flash memory device to be used by the OTA Upgrade cluster. Information about the device must be provided, such as the device type and sector size.

If a custom or unsupported Flash memory device is used then user-defined callback functions must be provided to perform Flash memory read, write, erase and initialization operations (if an NXP-supported device is used, standard callback functions will be used):

- A general set of functions (for use by all software components) can be specified through *pvFlashTable*.

- Optionally, an additional set of functions specifically for use by the OTA Upgrade cluster can be specified in the structure referenced by *psNvmStruct*.

This function must be called after the OTA Upgrade cluster has been created (after **eOTA_Create()** has been called either directly or indirectly) and before any other OTA Upgrade functions are called.

Parameters

- *pvFlashTable*: Pointer to general set of callback functions to perform Flash memory read, write, erase and initialization operations. If using an NXP-supported Flash memory device, set a null pointer to use standard callback functions
- *psNvmStruct*: Pointer to structure containing information on Flash memory device - see Section 49.11.4

Returns

- None

Parent topic: General Functions

eOTA_AllocateEndpointOTASpace

```
teZCL_Status eOTA_AllocateEndpointOTASpace(
    uint8 u8Endpoint,
    uint8 *pu8Data,
    uint8 u8NumberOfImages,
    uint8 u8MaxSectorsPerImage,
    bool_t bIsServer,
    uint8 *pu8CAPublicKey);
```

Description This function is used to allocate Flash memory space to store application images as part of the OTA upgrade process for the specified endpoint. The maximum number of images that are held at any one time must be specified as well the Flash memory start sector of every image. The maximum number of sectors used to store an image must also be specified.

The start sectors of the image space allocations are provided in an array. The index of an element of this array will subsequently be used to identify the stored image in other function calls.

Advice about the allocation of internal Flash memory space to OTA upgrade images on the client is provided in Appendix E.2

Parameters

- *u8Endpoint*: Number of endpoint for which Flash memory space is to be allocated
- *pu8Data*: Pointer to array containing the Flash memory start sector of each image (array index identifies image)
- *u8NumberOfImages*: Maximum number of application images that are stored in Flash memory at any one time
- *u8MaxSectorsPerImage*: Maximum number of sectors to be used to store an individual application image
- *bIsServer*: Side of cluster implemented on this device:
- TRUE - Server

- FALSE - Client
- *pu8CAPublicKey*: Pointer to Certificate Authority public key (provided in the security certificate from a company such as Certicom)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_INVALID_VALUE
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: General Functions

vOTA_GenerateHash

```
void vOTA_GenerateHash(
    tsZCL_EndPointDefinition *psEndPointDefinition,
    tsOTA_Common *psCustomData,
    bool bIsServer,
    bool bHeaderPresent,
    AESSW_Block_u *puHash,
    uint8 u8ImageLocation);
```

Description This function can be used to generate a hash checksum for an application image in Flash memory, using the Matyas-Meyer-Oseas cryptographic hash.

Parameters

- *psEndPointDefinition* Pointer to structure which defines endpoint corresponding to the application (see Section 6.1.1)
- *psCustomData* Pointer to data structure connected with event associated with the checksum (see Section 49.11.2)
- *bIsServer* Side of cluster implemented on this device:
- TRUE - Server
- FALSE - Client
- *bHeaderPresent* Presence of image header:
- TRUE - Present
- FALSE - Absent
- *puHash* Pointer to structure to receive calculated hash checksum
- *u8ImageLocation* Number of sector where image starts in Flash memory

Returns

- None

Parent topic: General Functions

eOTA_GetCurrentOtaHeader

```
teZCL_Status eOTA_GetCurrentOtaHeader(
    uint8 u8Endpoint,
    bool_t bIsServer,
    tsOTA_ImageHeader *psOTAHeader);
```

Description This function can be used to obtain the OTA header of the application image which is currently running on the local node.

The obtained parameter values are received in a tsOTA_ImageHeader structure.

Parameters

- *u8Endpoint* Number of endpoint on which cluster operates
- *bIsServer* Side of the cluster implemented on this device:
- TRUE - Server
- FALSE - Client
- *psOTAHeader* Pointer to structure to receive the current OTA header (see Section 49.11.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_PARAMETER_NULL

Parent topic: General Functions

Parent topic: Functions

Server Functions The following OTA Upgrade cluster functions can be used on the cluster server only:

1. eOTA_SetServerAuthorisation
2. eOTA_SetServerParams
3. eOTA_GetServerData
4. eOTA_EraseFlashSectorsForNewImage
5. eOTA_FlashWriteNewImageBlock
6. eOTA_NewImageLoaded
7. eOTA_ServerImageNotify
8. eOTA_ServerQueryNextImageResponse
9. eOTA_ServerImageBlockResponse
10. eOTA_SetWaitForDataParams
11. eOTA_ServerUpgradeEndResponse
12. eOTA_ServerSwitchToNewImage

13. eOTA_InvalidateStoredImage
14. eOTA_ServerQuerySpecificFileResponse

eOTA_SetServerAuthorisation

```
teZCL_Status eOTA_SetServerAuthorisation(
    uint8 u8Endpoint,
    eOTA_AuthorisationState eState,
    uint64 *pu64WhiteList,
    uint8 u8Size);
```

Description This function can be used to define a set of clients to which the server is authorized to download application images. The function allows all clients to be authorized or a list of selected authorized clients to be provided. Clients are specified in this list by means of their 64-bit IEEE/MAC addresses.

Parameters

- *u8Endpoint*: Number of endpoint (on server) on which cluster operates
- *eState*: Indicates whether a list of authorized clients is used or all clients are authorized - one of:
  - E_CLD_OTA_STATE_USE_LIST
  - E_CLD_OTA_STATE_ALLOW_ALL
- *pu64WhiteList*: Pointer to list of IEEE/MAC addresses of authorized clients (ignored if all clients are authorized through *eState* parameter)
- *u8Size* Number of clients in list
- (ignored if all clients are authorized through *eState* parameter)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Server Functions

eOTA_SetServerParams

```
teZCL_Status eOTA_SetServerParams(
    uint8 u8Endpoint,
    uint8 u8ImageIndex,
    tsCLD_PR_Ota *psOTAData);
```

Description This function can be used to set server parameter values (including query jitter, data size, image data, current time and upgrade time) for a particular image stored on the server. The parameter values to be set are specified in a structure, described in Section 49.11.22. For detailed descriptions of these parameters, refer to the *ZigBee Over-the-Air Upgrading Cluster Specification (095264)*from the ZigBee Alliance.

If this function is not called, default values are used for these parameters.

The current values of these parameters can be obtained using the function **eOTA_GetServerData()**.

The index of the image for which server parameter values are to be set must be specified. For an image stored in Flash memory, this index will take a value in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1).

Parameters

- *u8Endpoint*: Number of endpoint (on server) on which cluster operates
- *u8ImageIndex*: Index number of image
- *psOTAData*: Pointer to structure containing parameter values to be set (see Section 49.11.22)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Server Functions

eOTA_GetServerData

```
teZCL_Status eOTA_GetServerData(
    uint8 u8Endpoint,
    uint8 u8ImageIndex,
    tsCLD_PR_Ota *psOTAData);
```

Description This function can be used to obtain server parameter values (including query jitter, data size, image data, current time and upgrade time). The obtained parameter values are received in a structure, described in Section 49.11.22. For detailed descriptions of these parameters, refer to the *ZigBee Over-the-Air Upgrading Cluster Specification (095264)*from the ZigBee Alliance.

The values of these parameters can be set by the application using the function **eOTA_SetServerParams()**.

The index of the image for which server parameter values are to be obtained must be specified. For an image stored in the Flash memory, this index will take a value in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1).

Parameters

- *u8Endpoint*: Number of endpoint (on server) on which cluster operates
- *u8ImageIndex*: Index number of image
- *psOTAData*: Pointer to structure to receive parameter values (see Section 49.11.22)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Server Functions

eOTA_EraseFlashSectorsForNewImage

```
teZCL_Status eOTA_EraseFlashSectorsForNewImage(
    uint8 u8Endpoint,
    uint8 u8ImageIndex);
```

Description This function can be used to erase certain sectors of the Flash memory of the device in the OTA server node. The sectors allocated to the specified image index number will be erased so that the sectors (and index number) can be re-used. The function is normally called before writing a new upgrade image to Flash memory.

The specified image index number must be in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1).

Parameters

- *u8Endpoint*: Number of endpoint (on server) on which cluster operates
- *u8ImageIndex*: Index number of image

Returns

- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_SUCCESS

Parent topic: Server Functions

eOTA_FlashWriteNewImageBlock

```
teZCL_Status eOTA_FlashWriteNewImageBlock(
    uint8 u8Endpoint,
    uint8 u8ImageIndex,
    bool bIsServerImage,
    uint8 *pu8UpgradeBlockData,
    uint8 u8UpgradeBlockDataLength,
    uint32 u32FileOffSet);
```

Description This function can be used to write a block of an upgrade image to the devices internal Flash memory in the OTA server node. The image may be either of the following:

- An upgrade image for the server itself (the server will later be rebooted from this image)
- An upgrade image for one or more clients, which will later be made available for OTA distribution through the wireless network

The image in Flash memory to which the block belongs is identified by its index number. The specified image index number must be in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1).

Note that for JN518x, K32W041, and K32W061, internal Flash memory, writes must be 512-byte aligned.

Parameters

- *u8Endpoint*: Number of endpoint (on server) on which cluster operates
- *u8ImageIndex*: Index number of image
- *bIsServerImage*: Indicates whether new image is for the server or a client:
- TRUE - Server image
- FALSE - Client image
- *pu8UpgradeBlockData*: Pointer to image block to be written
- *u8UpgradeBlockDataLength*: Size, in bytes, of image block to be written
- *u32FileOffSet*: Offset of block from start of image file (in terms of number of bytes)

Returns

- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_FAIL
- E_ZCL_SUCCESS

Parent topic: Server Functions

eOTA_NewImageLoaded

```
teZCL_Status eOTA_NewImageLoaded(
        uint8 u8Endpoint,
        bool bIsImageOnCoProcessorMedia,
        tsOTA_CoProcessorOTAHeader
    *psOTA_CoProcessorOTAHeader);
```

Description This function can be used for two purposes which relate to a new application image and which depend on whether the image has been stored in the internal Flash memory of the device or in the external storage device of a co-processor (if any) within the server node:

- For an image stored in internal Flash memory, the function can be used to notify the OTA Upgrade cluster server on the specified endpoint that a new application image has been loaded into Flash memory and is available for download to clients. The server then validates the new image.
- For one or more images stored in the co-processor's external storage device, the function can be used to provide OTA header information for the image(s) to the cluster server. In the case of more than one image stored in co-processor storage, this function may replicate OTA header information for older images already registered with the server.

Note: The co-processor option is currently not supported for JN518x, K32W041, or K32W061.

Parameters

- *u8Endpoint*: Number of endpoint (on server) on which cluster operates
- *bIsImageOnCoProcessorMedia*: Flag indicating whether image is stored in co-processor external storage device:
- TRUE - Stored in co-processor

- FALSE - Stored in internal Flash memory
- *psOTA_CoProcessorOTAHeader*: Pointer to OTA headers of images which are held in co-processor storage device

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Server Functions

eOTA_ServerImageNotify

```
teZCL_Status eOTA_ServerImageNotify(
    uint8 u8SourceEndpoint,
    uint8 u8DestinationEndpoint,
    tsZCL_Address *psDestinationAddress,
    tsOTA_ImageNotifyCommand *psImageNotifyCommand);
```

Description This function issues an Image Notify message to one or more clients to indicate that a new application image is available for download.

The message can be unicast to an individual client or multicast to selected clients (but cannot be broadcast to all clients, for security reasons).

Parameters

- *u8SourceEndpoint*: Number of endpoint (on server) from which the message is sent
- *u8DestinationEndpoint*: Number of endpoint (on client) to which the message is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target client for the message - a multicast to more than one client is also possible (see Section 6.1.4)
- *psImageNotifyCommand*: Pointer to structure containing payload for message (see Section 49.11.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Server Functions

eOTA_ServerQueryNextImageResponse

```
teZCL_Status eOTA_ServerQueryNextImageResponse(
        uint8 u8SourceEndpoint,
    uint8 u8DestinationEndpoint,
    tsZCL_Address *psDestinationAddress,
    tsOTA_QueryImageResponse
    *psQueryImageResponsePayload,
    uint8 u8TransactionSequenceNumber);
```

Description This function issues a Query Next Image Response to a client which has sent a Query Next Image Request (the arrival of this request triggers the event E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_REQUEST on the server).

The Query Next Image Response contains information on the latest application image available for download to the client, including the image size and file version.

Note: The cluster server responds automatically to a Query Next Image Request, so it is not normally necessary for the application to call this function.

You are also required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndpoint*: Number of endpoint (on server) from which the response is sent
- *u8DestinationEndpoint*: Number of endpoint (on client) to which the response is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target client for the response (see Section 6.1.4)
- *psQueryImageResponsePayload*: Pointer to structure containing payload for response (see Section 49.11.7)
- *u8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Server Functions

eOTA_ServerImageBlockResponse

```
teZCL_Status eOTA_ServerImageBlockResponse(
    uint8 u8SourceEndpoint,
    uint8 u8DestinationEndpoint,
    tsZCL_Address *psDestinationAddress,
    tsOTA_ImageBlockResponsePayload
    *psImageBlockResponsePayload,
    uint8 u8BlockSize,
    uint8 u8TransactionSequenceNumber);
```

Description This function issues an Image Block Response, containing a block of image data, to a client to which the server is downloading an application image. The function is called after receiving an Image Block Request from the client, indicating that the client is ready to receive the next block of the application image (the arrival of this request triggers the event E_CLD_OTA_COMMAND_BLOCK_REQUEST on the server).

The size of the block, in bytes, is specified as part of the function call. This must be less than or equal to the maximum possible block size defined in the **zcl_options.h** file (see Section 49.13).

Note: The cluster server responds automatically to an Image Block Request, so it is not normally necessary for the application to call this function.

You are also required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndpoint*: Number of endpoint (on server) from which the response is sent
- *u8DestinationEndpoint*: Number of endpoint (on client) to which the response is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target client for the response (see Section 6.1.4)
- *psImageBlockResponsePayload*: Pointer to structure containing payload for response (see Section 49.11.10)
- *u8BlockSize*: Size, in bytes, of block to be transferred
- *u8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Server Functions

eOTA_SetWaitForDataParams

```
teZCL_Status eOTA_SetWaitForDataParams(
    uint8 u8Endpoint,
    uint16 u16ClientAddress,
    tsOTA_WaitForData *sWaitForDataParams);
```

Description This function can be used to send an Image Block Response with a status of OTA_STATUS_WAIT_FOR_DATA to a client, in response to an Image Block Request from the client.

The payload of this response includes a new value for the 'block request delay' attribute on the client. This value can be used by the client for 'rate limiting' -that is, to control the rate at which the client requests data blocks from the server and therefore the average OTA download rate from the server to the client.

Rate limiting is described in more detail in Section 49.8.1.

Parameters

- *u8Endpoint*: Number of endpoint (on server) from which the response is sent
- *u16ClientAddress*: Network address of client device to which the response is sent
- *sWaitForDataParams*: Pointer to structure containing 'Wait for Data' parameter values for Image Block Response payload (see Section49.11.15)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Server Functions

eOTA_ServerUpgradeEndResponse

```
teZCL_Status eOTA_ServerUpgradeEndResponse(
    uint8 u8SourceEndpoint,
    uint8 u8DestinationEndpoint,
    tsZCL_Address *psDestinationAddress,
    tsOTA_UpgradeEndResponsePayload
    *psUpgradeResponsePayload,
    uint8 u8TransactionSequenceNumber);
```

Description This function issues an Upgrade End Response to a client to which the server has been downloading an application image. The function is called after receiving an Upgrade End Request from the client, indicating that the client has received the entire application image and verified it (the arrival of this request triggers the event E_CLD_OTA_COMMAND_UPGRADE_END_REQUEST on the server).

The Upgrade End Response includes the upgrade time for the downloaded image as well as the current time (the client will use this information to implement a delay before upgrading the running application image).

Note: The cluster server responds automatically to an Upgrade End Request, so it is not normally necessary for the application to call this function.

You are also required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndpoint*: Number of endpoint (on server) from which the response is sent
- *u8DestinationEndpoint*: Number of endpoint (on client) to which the response is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target client for the response (see Section 6.1.4)
- *psUpgradeResponsePayload*: Pointer to structure containing payload for response (see Section 49.11.12)
- *u8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Server Functions

eOTA_ServerSwitchToNewImage

```
teZCL_Status eOTA_ServerSwitchToNewImage(
    uint8 u8Endpoint,
    uint8 u8ImageIndex);
```

Description This function can be used to force a reset of the device in the OTA server node and, on reboot, run a new application image that has been saved in the attached Flash memory.

Before forcing the reset of the remove device, the function checks whether the version of the new image is greater than the version of the current image. If this is the case, the function invalidates the currently running image in Flash memory and initiates a software reset - otherwise, it returns an error.

The new application image is identified by its index number. The specified image index number must be in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1).

Parameters

- *u8Endpoint*: Number of endpoint (on server) on which cluster operates
- *u8ImageIndex*: Index number of image

Returns

- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_FAIL
- E_ZCL_SUCCESS

Parent topic: Server Functions

eOTA_InvalidateStoredImage

```
teZCL_Status eOTA_InvalidateStoredImage(
    uint8 u8Endpoint,
    uint8 u8ImageIndex);
```

Description This function can be used to invalidate an application image that is held in the Flash memory of the device. Once the image has been invalidated, it will no longer to available for OTA upgrade.

The image to be invalidated is identified by its index number. The specified image index number must be in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1).

Parameters

- *u8Endpoint*: Number of endpoint (on server) on which cluster operates
- *u8ImageIndex*: Index number of image to be invalidated

Returns

- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_SUCCESS

Parent topic: Server Functions

eOTA_ServerQuerySpecificFileResponse

```
teZCL_Status eOTA_ServerQuerySpecificFileResponse(
    uint8 u8SourceEndpoint,
    uint8 u8DestinationEndpoint,
    tsZCL_Address *psDestinationAddress,
    tsOTA_QuerySpecificFileResponsePayload
    *psQuerySpecificFileResponsePayload,
    uint8 u8TransactionSequenceNumber);
```

Description This function can be used to issue a Query Specific File Response to a client which has sent a Query Specific File Request (the arrival of this request triggers the event E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_REQUEST on the server). The Query Specific File Response contains information on the latest device-specific file available for download to the client, including the file size and file version.

You are also required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

- *u8SourceEndpoint*: Number of endpoint (on server) from which the response is sent
- *u8DestinationEndpoint*: Number of endpoint (on client) to which the response is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target client
- *psQuerySpecificFileResponsePayload*:
- Pointer to structure containing payload for Query Specific File Response (see Section 49.11.20)
- *u8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Server Functions

Parent topic: Functions

Client Functions The following OTA Upgrade cluster functions can be used on the cluster client only:

1. eOTA_SetServerAddress
2. eOTA_ClientQueryNextImageRequest
3. eOTA_ClientImageBlockRequest
4. eOTA_ClientImagePageRequest
5. eOTA_ClientUpgradeEndRequest
6. eOTA_HandleImageVerification
7. eOTA_UpdateClientAttributes
8. eOTA_RestoreClientData
9. vOTA_SetImageValidityFlag
10. eOTA_ClientQuerySpecificFileRequest
11. eOTA_SpecificFileUpgradeEndRequest
12. vOTA_SetLowVoltageFlag

eOTA_SetServerAddress

```
teZCL_Status eOTA_SetServerAddress(
    uint8 u8Endpoint,
    uint64 u64IeeeAddress,
    uint16 u16ShortAddress);
```

Description This function sets the addresses (64-bit IEEE/MAC address and 16-bit network address) of the OTA Upgrade cluster server that will be used to provide application upgrade images to the local client.

The function should be called after a server discovery has been performed to find a suitable server - this is done by sending out a Match Descriptor Request using the function **ZPS_eAplZdpMatchDescRequest()** described in the*ZigBee 3.0 Stack User Guide (JNUG3130)*. The server discovery must be completed and a server address set before any OTA-related message exchanges can occur (e.g. image request).

Parameters

- *u8Endpoint*: Number of endpoint corresponding to application
- *u64IeeeAddress*: IEEE/MAC address of server
- *u16ShortAddress*: Network address of server

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Client Functions

eOTA_ClientQueryNextImageRequest

```
teZCL_Status eOTA_ClientQueryNextImageRequest(
    uint8 u8SourceEndpoint,
    uint8 u8DestinationEndpoint,
    tsZCL_Address *psDestinationAddress,
    tsOTA_QueryImageRequest
*psQueryImageRequest);
```

Description This function issues a Query Next Image Request to the server and should be called in either of the following situations:

- to poll for a new application image (typically used in this way by an End Device) - in this case, the function should normally be called periodically
- to respond to an Image Notify message from the server, which indicated that a new application image is available for download - in this case, the function call should be prompted by the event E_CLD_OTA_COMMAND_IMAGE_NOTIFY

The payload of the request includes the relevant image type, current file version, hardware version and manufacturer code.

As a result of this function call, a Query Next Image Response will (eventually) be received from the server. The arrival of this response will trigger an E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE event.

Parameters

- *u8SourceEndpoint*: Number of endpoint (on client) from which the request is sent
- *u8DestinationEndpoint*: Number of endpoint (on server) to which the request is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target server (see Section 6.1.4)
- *psQueryImageRequest*: Pointer to structure containing payload for request (see Section 49.11.6)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Client Functions

eOTA_ClientImageBlockRequest

```
teZCL_Status eOTA_ClientImageBlockRequest(
        uint8 u8SourceEndpoint,
        uint8 u8DestinationEndpoint,
        tsZCL_Address *psDestinationAddress,
        tsOTA_BlockRequest
    *psOtaBlockRequest);
```

Description This function can be used during an image download to send an Image Block Request to the server, in order to request the next block of image data.

As a result of this function call, an Image Block Response containing the requested data block will (eventually) be received from the server. The arrival of this response will trigger an E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE event.

Note: The cluster client automatically sends Image Block Requests to the server during a download, so it is not normally necessary for the application to call this function.

Parameters

- *u8SourceEndpoint*: Number of endpoint (on client) from which the request is sent
- *u8DestinationEndpoint*: Number of endpoint (on server) to which the request is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target server (see Section 6.1.4)
- *psOtaBlockRequest*: Pointer to structure containing payload for request (see Section 49.11.8)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Client Functions

eOTA_ClientImagePageRequest

```
teZCL_Status eOTA_ClientImagePageRequest(
    uint8 u8SourceEndpoint,
    uint8 u8DestinationEndpoint,
    tsZCL_Address *psDestinationAddress,
    tsOTA_ImagePageRequest *psOtaPageRequest);
```

Description This function can be used during an image download to send an Image Page Request to the server, in order to request the next page of image data. In this function call, a structure must be supplied which contains the payload data for the request. This data includes the page size, in bytes.

Note 1: Image Page Requests can be used instead of Image Block Requests if page requests have been enabled in the **zcl_options.h** file for the client and server (see Section 49.13).

Note 2: The cluster client automatically sends Image Page Requests (if enabled) to the server during a download, so it is not normally necessary for the application to call this function.

As a result of this function call, a sequence of Image Block Responses containing the requested data will (eventually) be received from the server. The arrival of each response will trigger an E_CLD_OTA_COMMAND_BLOCK_RESPONSE event on the client. If this function is used (rather than the stack) to issue Image Page Requests, it is the responsibility of the application to keep a count of the number of data bytes received since the Image Page Request was issued - when all the requested page data has been received, this count will equal the specified page size.

Page requests are described in more detail in Section 49.8.4.

Parameters

- *u8SourceEndpoint*: Number of endpoint (on client) from which the request is sent
- *u8DestinationEndpoint*: Number of endpoint (on server) to which the request is sent

- *psDestinationAddress*: Pointer to structure containing the address of the target server (see Section 6.1.4)
- *psOtaPageRequest*: Pointer to structure containing payload for request (see Section 49.11.9)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Client Functions

eOTA_ClientUpgradeEndRequest

```
teZCL_Status eOTA_ClientUpgradeEndRequest(
    uint8 u8SourceEndpoint,
    uint8 u8DestinationEndpoint,
    tsZCL_Address *psDestinationAddress,
    tsOTA_UpgradeEndRequestPayload
    *psUpgradeEndRequestPayload);
```

Description This function can be used during an image download to send an Upgrade End Request to the server. This is normally used to indicate that all the image data has been received and that the image has been successfully verified - it is the responsibility of the client to determine when all the image data has been received (using the image size quoted in the original Query Next Image Response) and then to verify the image.

In addition to the status OTA_STATUS_SUCCESS described above, the function can be used by the client to report other conditions to the server:

- OTA_REQUIRE_MORE_IMAGE: The downloaded image was successfully received and verified, but the client requires multiple images before performing an upgrade
- OTA_STATUS_INVALID_IMAGE: The downloaded image failed the verification checks and will be discarded
- OTA_STATUS_ABORT The image download that is currently in progress should be cancelled

In all three of the above cases, the client may then request another download.

When the function is called to report success, an Upgrade End Response will (eventually) be received from the server, indicating when the image upgrade should be implemented (a time delay may be indicated in the response). The arrival of this response will trigger an E_CLD_OTA_COMMAND_UPGRADE_END_RESPONSE event.

Note: The cluster client automatically sends an Upgrade End Request to the server on completion of a download, so it is not normally necessary for the application to call this function.

Parameters

- *u8SourceEndpoint*: Number of endpoint (on client) from which the request is sent
- *u8DestinationEndpoint*: Number of endpoint (on server) to which the request is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target server (see Section 6.1.4)
- *psUpgradeEndRequestPayload*: Pointer to structure containing payload for request, including reported status (see Section49.11.11)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Client Functions

eOTA_HandleImageVerification

```
teZCL_Status eOTA_HandleImageVerification(
    uint8 u8SourceEndPointId,
    uint8 u8DstEndpoint,
    teZCL_Status eImageVerificationStatus);
```

Description This function transmits an upgrade end request with the specified status.

Parameters

- *u8SourceEndPointId*: Identifier of endpoint on which the cluster client operates
- *u8DstEndpoint*: Identifier of endpoint (on the server) to which the upgrade end request is sent
- *eImageVerificationStatus*: Image status code

Returns

- E_ZCL_FAIL
- E_ZCL_SUCCESS

Parent topic: Client Functions

eOTA_UpdateClientAttributes

```
teZCL_Status eOTA_UpdateClientAttributes(
    uint8 u8Endpoint);
```

Description This function can be used on a client to set the OTA Upgrade cluster attributes to their default values. It should be called during application initialization after the cluster instance has been created using **eOTA_Create()**.

Following subsequent resets, provided that context data has been saved, the application should call **eOTA_RestoreClientData()** instead of this function.

Parameters

- *u8Endpoint*: Number of endpoint corresponding to context data

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Client Functions

eOTA_RestoreClientData

```
teZCL_Status eOTA_RestoreClientData(
    uint8 u8Endpoint,
    tsOTA_PersistedData *psOTAData,
    bool_t bReset);
```

Description This function can be used to restore OTA Upgrade context data that has been previously saved to Flash memory (using the NVM) on the local client - for example, it restores the OTA Upgrade attribute values. The function can be used to restore the data in RAM following a device reset or simply to refresh the data in RAM.

Parameters

- *8Endpoint*: Number of endpoint corresponding to context data
- *psOTAData*: Pointer to structure containing the context data to be restored (see Section 49.11.13)
- *bReset*: Indicates whether the data restoration follows a reset:
- TRUE - Follows a reset
- FALSE - Does not follow a reset

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Client Functions

vOTA_SetImageValidityFlag

```
void vOTA_SetImageValidityFlag(
    uint8 u8Location,
    tsOTA_Common *psCustomData,
    bool bSet,
    tsZCL_EndPointDefinition *psEndPointDefinition);
```

Description This function can be used to set an image validity flag once a downloaded upgrade image has been received and verified by the client.

Parameters

- *u8Location*: Number of sector where image starts in Flash memory
- *psCustomData*: Pointer to custom data for image (see Section 49.11.2)
- *bSet*: Flag state to be set:
- TRUE - Reset
- FALSE - No reset
- *psEndPointDefinition*: Pointer to endpoint definition (see Section 6.1.1)

Returns

- None

Parent topic: Client Functions

eOTA_ClientQuerySpecificFileRequest

```
eOTA_ClientQuerySpecificFileRequest(
    uint8 u8SourceEndpoint,
    uint8 u8DestinationEndpoint,
    tsZCL_Address *psDestinationAddress,
    tsOTA_QuerySpecificFileRequestPayload
*psQuerySpecificFileRequestPayload);
```

Description This function can be used to issue a Query Specific File Request to the server. It should be called to request a device-specific file from the server. As a result of this function call, a Query Specific File Response will (eventually) be received in reply.

Parameters

- *u8SourceEndpoint*: Number of endpoint (on client) from which the request is sent
- *u8DestinationEndpoint*: Number of endpoint (on server) to which the request is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target server
- *psQuerySpecificFileRequestPayload*: Pointer to structure containing payload for Query Specific File Request

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Client Functions

eOTA_SpecificFileUpgradeEndRequest

```
eOTA_SpecificFileUpgradeEndRequest(
    uint8 u8SourceEndPointId,
    uint8 u8Status);
```

Description This function can be used to issue an Upgrade End Request for the device-specific file download that is in progress in order to indicate to the server that the download has completed. This request can be issued by the client optionally after the downloaded image has been verified and found to be valid.

Parameters

- *u8SourceEndPointId*: Number of endpoint (on client) from which the request is sent
- *u8Status*: Download status of device-specific file - if the file has been completely and successfully received, this parameter must be set to OTA_STATUS_SUCCESS

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

Parent topic: Client Functions

vOTA_SetLowVoltageFlag

```
void vOTA_SetLowVoltageFlag(bool bValue);
```

Description This function can be used to configure the low-voltage flag on a node hosting an OTA Upgrade cluster client. This flag should be set when the supply voltage to the underlying hardware is below that required for normal operation and the node should not participate in an OTA upgrade.

- When the flag is set, the client stops sending Image Block Requests to the server
- When the flag is cleared, the client resumes sending Image Block Requests to the server

Use of the low-voltage flag must be enabled at compile-time by including the macro OTA_UPGRADE_VOLTAGE_CHECK in the **zcl_options.h** file.

Use of the low-voltage flag is described further in Section 49.8.7.

Parameters

- *bValue*: Determines the state of the low-voltage flag, as follows:
  - TRUE - Sets the flag
  - FALSE - Clears the flag

Returns

- None

Parent topic: Client Functions

Parent topic: Functions

Parent topic:*OTA Upgrade cluster*

Structures

tsOTA_ImageHeader The following structure contains information for the OTA header:

```
typedef struct
{
        uint32 u32FileIdentifier;
    uint16 u16HeaderVersion;
    uint16 u16HeaderLength;
    uint16 u16HeaderControlField;
    uint16 u16ManufacturerCode;
    uint16 u16ImageType;
    uint32 u32FileVersion;
    uint16 u16StackVersion;
    uint8  stHeaderString[OTA_HEADER_STRING_SIZE];
```

(continues on next page)

(continued from previous page)

```
        uint32 u32TotalImage;
    uint8  u8SecurityCredVersion;
    uint64 u64UpgradeFileDest;
    uint16 u16MinimumHwVersion;
    uint16 u16MaxHwVersion;
}tsOTA_ImageHeader;
```

where:

- u32FileIdentifier is a 4-byte value equal to 0x0BEEF11E which indicates that the file contains an OTA upgrade image
- u16HeaderVersion is the version of the OTA header expressed as a 2-byte value in which the most significant byte contains the major version number and the least significant byte contains the minor version number
- u16HeaderLength is the full length of the OTA header, in bytes
- u16HeaderControlField is a bitmap indicating certain information about the file, as detailed in table below.

Bit	Information
0	Security credential version (in OTA header):

1: Field present in header

0: Field not present in header

| |1|Device-specific file (also see u64UpgradeFileDest):

1: File is device-specific

0: File is not device-specific

| |2|Maximum and minimum hardware version (in OTA header):

1: Field present in header

0: Field not present in header

| |3-15|Reserved|

- u16ManufacturerCode is the ZigBee-assigned manufacturer code (0xFFFF is a wildcard value, representing any manufacturer)
- u16ImageType is a unique value representing the image type, where this value is normally manufacturer-specific but certain values have been reserved for specific file types, as indicated below (the wildcard value of 0xFFFF represents any file type):

Value	File Type
0x0000 – 0xFFBF	Manufacturer-specific
0xFFC0	Security credential
0xFFC1	Configuration
0xFFC2	Log
0xFFC3 – 0xFFFE	Reserved
0xFFFF	Wildcard

- u32FileVersion contains the release and build numbers of the application and stack used to produce the application image - for details of the file version format, refer to the *ZigBee Over-the-Air Upgrading Cluster Specification (095264)*

- u16StackVersion contains ZigBee stack version that is used by the application (this is 0x0002 for ZigBee PRO)
- stHeaderString[] is a manufacturer-specific string that can be used to store any useful human-readable information
- u32TotalImage is the total size, in bytes, of the image that will be transferred over-the air (including the OTA header and any optional data)
- u8SecurityCredVersion indicates the security credential version type that is required by the client in order to install the image - the possibilities are SE1.0 (0x0), SE1.1 (0x1) and SE2.0 (0x2)
- u64UpgradeFileDest contains the IEEE/MAC address of the destination device for the file, in the case when the file is device-specific (as indicated by bit 1 of u16HeaderControlField)
- u16MinimumHwVersion indicates the earliest hardware platform on which the image should be used, expressed as a 2-byte value in which the most significant byte contains the hardware version number and the least significant byte contains the revision number
- u16MaxHwVersion indicates the latest hardware platform on which the image should be used, expressed as a 2-byte value in which the most significant byte contains the hardware version number and the least significant byte contains the revision number

Parent topic: Structures

tsOTA_Common The following structure contains data relating to an OTA message received by the cluster (server or client) - this data is used for callback functions and the local OTA state machine:

```
typedef struct
{
    tsZCL_ReceiveEventAddress   sReceiveEventAddress;
    tsZCL_CallBackEvent         sOTACustomCallBackEvent;
    tsOTA_CallBackMessage       sOTACallBackMessage;
} tsOTA_Common;
```

The fields are for internal use and no knowledge of them is required. The tsOTA_CallBackMessage structure is described in Section 49.11.21.

Parent topic: Structures

tsOTA_HwFncTable The following structure contains pointers to callback functions to be used by the OTA Upgrade cluster to perform initialization, erase, write and read operations on Flash memory (if these functions are not specified, standard NXP functions will be used):

```
typedef struct
{
     void (*prInitHwCb)(uint8, void*);
     void (*prEraseCb) (uint8 u8Sector);
     void (*prWriteCb) (uint32 u32FlashByteLocation,
                        uint16 u16Len,
                        uint8 *pu8Data);
     void (*prReadCb)  (uint32 u32FlashByteLocation,
                        uint16 u16Len,
                        uint8 *pu8Data);
} tsOTA_HwFncTable;
```

where:

- prInitHwCb is a pointer to a callback function that is called after a cold or warm start to perform any initialization required for the Flash memory device
- prEraseCb is a pointer to a callback function that is called to erase a specified sector of Flash memory
- prWriteCb is a pointer to a callback function that is called to write a block of data to a sector, starting the write at a specified byte location in the sector (address zero is the start of the sector)
- prReadCb is a pointer to a callback function that is called to read a block of data from a sector, starting the read at a specified byte location in the sector (address zero is the start of the sector)

Parent topic: Structures

tsNvmDefs The following structure contains information used to configure access to Flash memory:

```
typedef struct
{
    tsOTA_HwFncTable sOtaFnTable;
    uint32           u32SectorSize;
    uint8            u8FlashDeviceType;
}tsNvmDefs;
```

where:

- sOtaFnTable is a structure specifying the callback functions to be used by the cluster to perform initialization, erase, write and read operations on the Flash memory device (see Section 49.11.3) - if user-defined callback functions are not specified, standard NXP functions will be used
- u32SectorSize is the size of a sector of the Flash memory device, in bytes
- u8FlashDeviceType is a value indicating the type of Flash memory device, one of:
  - E_FL_CHIP_INTERNAL (Device internal Flash- default)

Parent topic: Structures

tsOTA_ImageNotifyCommand The following structure contains the payload data for an Image Notify message issued by the server when a new upgrade image is available for download:

```
typedef struct
{
        teOTA_ImageNotifyPayloadType ePayloadType;
        uint32                       u32NewFileVersion;
        uint16                       u16ImageType;
        uint16                       u16ManufacturerCode;
        uint8                        u8QueryJitter;
}tsOTA_ImageNotifyCommand;
```

where:

- ePayloadType is a value indicating the type of payload of the command (enumerations are available - see Section 49.12.4)
- u32NewFileVersion is the file version of the client upgrade image that is currently available for download (the wild card of 0xFFFFFFFF is used to indicate that all clients should upgrade to this image)

- u16ImageType is a number indicating the type of image that is available for download (the wild card of 0xFFFF is used to indicate that all image types are involved)
- u16ManufacturerCode is a ZigBee-assigned number identifying the manufacturer to which the available image is connected (if all manufacturers are involved, this value should not be set)
- u8QueryJitter is a value between 1 and 100 (inclusive) which is used by the receiving client to decide whether to reply to this Image Notify message - for information on 'Query Jitter', refer to Section 49.7

Parent topic: Structures

tsOTA_QueryImageRequest The following structure contains payload data for a Query Next Image Request issued by a client to poll the server for an upgrade image or to respond to an Image Notify message from the server:

```
typedef struct
{
    uint32 u32CurrentFileVersion;
    uint16 u16HardwareVersion;
    uint16 u16ImageType;
    uint16 u16ManufacturerCode;
    uint8  u8FieldControl;
}tsOTA_QueryImageRequest;
```

where:

- u32CurrentFileVersion is the file version of the application image that is currently running on the client that sent the request
- u16HardwareVersion is the hardware version of the client device (this information is optional - see u8FieldControl below)
- u16ImageType is a value in the range 0x0000-0xFFBF which identifies the type of image currently running on the client
- u16ManufacturerCode is the ZigBee-assigned number identifying the manufacturer of the client device
- u8FieldControl is a bitmap indicating whether certain optional information about the client is included in this Query Next Image Request message. Currently, this optional information consists only of the hardware version (contained in u16HardwareVersion above) - bit 0 is set to '1' if the hardware version is included or to '0' otherwise (all other bits are reserved)

Parent topic: Structures

tsOTA_QueryImageResponse The following structure contains payload data for a Query Next Image Response issued by the server (as the result of a Query Next Image Request from a client):

```
typedef struct
{
    uint32 u32ImageSize;
    uint32 u32FileVersion;
    uint16 u16ManufacturerCode;
    uint16 u16ImageType;
    uint8  u8Status;
}tsOTA_QueryImageResponse;
```

where:

- u32ImageSize is the total size of the available image, in bytes
- u32FileVersion is the file version of the available image
- u16ManufacturerCode is the manufacturer code that was received from the client in the Query Next Image Request message
- u16ImageType is the image type that was received from the client in the Query Next Image Request message
- u8Status indicates whether a suitable image is available for download:
  - OTA_STATUS_SUCCESS: A suitable image is available
  - OTA_STATUS_NO_IMAGE_AVAILABLE: No suitable image is available
    - The other elements of the structure are only included in the case of success.

Parent topic: Structures

tsOTA_BlockRequest The following structure contains payload data for an Image Block Request issued by a client to request an image data block from the server:

```
typedef struct
{
    uint64 u64RequestNodeAddress;
    uint32 u32FileOffset;
    uint32 u32FileVersion;
    uint16 u16ImageType;
    uint16 u16ManufactureCode;
    uint16 u16BlockRequestDelay;
    uint8 u8MaxDataSize;
    uint8 u8FieldControl;
}tsOTA_BlockRequest;
```

where:

- u64RequestNodeAddress is the IEEE/MAC address of the client device from which the request originates (this information is optional - see u8FieldControl below)
- u32FileOffset specifies the offset from the beginning of the upgrade image, in bytes, of the requested data block (this value is therefore determined by the amount of image data previously received)
- u32FileVersion is the file version of the upgrade image for which a data block is being requested
- u16ImageType is a value in the range 0x0000-0xFFBF which identifies the type of image for which a data block is being requested
- u16ManufactureCode is the ZigBee-assigned number identifying the manufacturer of the client device from which the request originates
- u16BlockRequestDelay is used in 'rate limiting' to specify the value of the 'block request delay' attribute for the client - this is minimum time, in milliseconds, that the client must wait between consecutive block requests (the client will update the local attribute with this value). If the server does not support rate limiting or does not need to limit the download rate to the client, this field will be set to 0
- u8MaxDataSize specifies the maximum size, in bytes, of the data block that the client can receive in one transfer (the server must therefore not send a data block that is larger than indicated by this value)

- u8FieldControl is a bitmap indicating whether certain optional information about the client is included in this Image Block Request message. Currently, this optional information consists only of the IEEE/MAC address of the client (contained in 64RequestNodeAddress above) - bit 0 is set to '1' if this address is included or to '0' otherwise (all other bits are reserved)

Parent topic: Structures

tsOTA_ImagePageRequest The following structure contains payload data for an Image Page Request issued by a client to request a page of image data (multiple blocks) from the server:

```
typedef struct
{
    uint64 u64RequestNodeAddress;
    uint32 u32FileOffset;
    uint32 u32FileVersion;
    uint16 u16PageSize;
    uint16 u16ResponseSpacing;
    uint16 u16ImageType;
    uint16 u16ManufactureCode;
    uint8 u8MaxDataSize;
    uint8 u8FieldControl;
}tsOTA_ImagePageRequest;
```

where:

- u64RequestNodeAddress is the IEEE/MAC address of the client device from which the request originates (this information is optional - see u8FieldControl below)
- u32FileOffset specifies the offset from the beginning of the upgrade image, in bytes, of the first data block of the requested page (this value is therefore determined by the amount of image data previously received)
- u32FileVersion is the file version of the upgrade image for which data is being requested
- u16PageSize is the total number of data bytes (in the page) to be returned by the server before the next Image Page Request can be issued (this must be larger than the value of u8MaxDataSize below)
- u16ResponseSpacing specifies the time-interval, in milliseconds, that the server should introduce between consecutive transmissions of Image Block Responses (which is sent in response to the Image Page Request)
- u16ImageType is a value in the range 0x0000-0xFFBF which identifies the type of image for which data is being requested
- u16ManufactureCode is the ZigBee-assigned number identifying the manufacturer of the client device from which the request originates
- u8MaxDataSize specifies the maximum size, in bytes, of the data block that the client can receive in one transfer (the server must therefore not send a data block in an Image Block Response that is larger than indicated by this value)
- u8FieldControl is a bitmap indicating whether certain optional information about the client is included in this Image Page Request message. Currently, this optional information consists only of the IEEE/MAC address of the client (contained in 64RequestNodeAddress above) - bit 0 is set to '1' if this address is included or to '0' otherwise (all other bits are reserved)

Parent topic: Structures

tsOTA_ImageBlockResponsePayload The following structure contains payload data for an Image Block Response issued by the server (as the result of an Image Block Request from a client):

```
typedef struct
{
        uint8 u8Status;
        union
        {
                tsOTA_WaitForData                sWaitForData;
      tsOTA_SuccessBlockResponsePayload   sBlockPayloadSuccess;
        }uMessage;
}tsOTA_ImageBlockResponsePayload;
```

where:

- u8Status indicates whether a data block is included in the response:
  - OTA_STATUS_SUCCESS: A data block is included
  - OTA_STATUS_WAIT_FOR_DATA: No data block is included - client should re-request a data block after a waiting time
- The element used from the union depends on the status reported above:
  - sWaitForData is a structure containing information used to instruct the requesting client to wait for a time before requesting the data block again or requesting the next data block (see Section 49.11.15) - this information is only provided in the case of the status OTA_STATUS_WAIT_FOR_DATA
  - sBlockPayloadSuccess is a structure containing a requested data block and associated information (see Section 49.11.13) - this data is only provided in the case of the status OTA_STATUS_SUCCESS

Parent topic: Structures

tsOTA_UpgradeEndRequestPayload The following structure contains payload data for an Upgrade End Request issued by a client to terminate/close an image download from the server:

```
typedef struct
{
    uint32 u32FileVersion;
    uint16 u16ImageType;
    uint16 u16ManufacturerCode;
        uint8  u8Status;
}tsOTA_UpgradeEndRequestPayload;
```

where:

- u32FileVersion is the file version of the upgrade image which has been downloaded
- u16ImageType is the type of the upgrade image which has been downloaded
- u16ManufacturerCode is the ZigBee-assigned number identifying the manufacturer of the client device from which the request originates
- u8Status is the reported status of the image download, one of:
  - OTA_STATUS_SUCCESS (successfully downloaded and verified)
  - OTA_STATUS_INVALID_IMAGE (downloaded but failed verification)
  - OTA_REQUIRE_MORE_IMAGE (other images needed)
  - OTA_STATUS_ABORT (download in progress is to be aborted)

Parent topic: Structures

tsOTA_UpgradeEndResponsePayload The following structure contains payload data for an Upgrade End Response issued by the server (as the result of an Upgrade End Request from a client):

```
typedef struct
{
    uint32 u32UpgradeTime;
    uint32 u32CurrentTime;
    uint32 u32FileVersion;
    uint16 u16ImageType;
    uint16 u16ManufacturerCode;
}tsOTA_UpgradeEndResponsePayload;
```

where:

- u32UpgradeTime is the UTC time, in seconds, at which the client should upgrade the running image with the downloaded image. If the server does not support UTC time (indicated by a zero value for u32CurrentTime), the client should interpret this value as a time delay before performing the image upgrade
- u32CurrentTime is the current UTC time, in seconds, on the server. If UTC time is not supported by the server, this value should be set to zero. If this value is set to 0xFFFFFFFF, this indicates that the client should wait for an upgrade command from the server before performing the image upgrade

Note: If the client does not support UTC time but both of the above time values are non-zero, the client will take the difference between the two times as a time delay before performing the image upgrade.

- u32FileVersion is the file version of the downloaded application image (a wild card value of 0xFFFFFFFF can be used when the same response is sent to client devices from different manufacturers)
- u16ImageType is the type of the downloaded application image (a wild card value of 0xFFFF can be used when the same response is sent to client devices from different manufacturers)
- u16ManufacturerCode is the manufacturer code that was received from the client in the Upgrade End Request message (a wild card value of 0xFFFF can be used when the same response is sent to client devices from different manufacturers)

Parent topic: Structures

tsOTA_SuccessBlockResponsePayload The following structure contains payload data for an Image Block Response which reports 'success' and therefore contains a block of image data (see Section 49.11.10):

```
typedef struct
{
    uint8* pu8Data;
           uint32 u32FileOffset;
           uint32 u32FileVersion;
           uint16 u16ImageType;
           uint16 u16ManufacturerCode;
    uint8  u8DataSize;
}tsOTA_SuccessBlockResponsePayload;
```

where:

- pu8Data is a pointer to the start of the data block being transferred
- u32FileOffset is the offset, in bytes, of the start of the data block from the start of the image (normally, the same offset as specified in the Image Block Request)

- u32FileVersion is the file version of the upgrade image to which the included data block belongs
- u16ImageType is the type of the upgrade image to which the included data block belongs
- u16ManufacturerCode is the manufacturer code that was received from the client in the Image Block Request
- u8DataSize is the length, in bytes, of the included data block (this must be less than or equal to the maximum data block length for the client, specified in the Image Block Request)

Parent topic: Structures

tsOTA_BlockResponseEvent The following structure contains payload data for an Image Block Response containing data other than upgrade image data.

```
typedef struct
{
    uint8    u8Status;
    uint8   *pu8Data;
    uint8    u8DataSize;
}tsOTA_BlockResponseEvent;
```

where:

- u8Status indicates whether a suitable upgrade image is available:
  - OTA_STATUS_SUCCESS: A suitable image is available
  - OTA_STATUS_NO_IMAGE_AVAILABLE: No suitable image is available
- pu8Data is a pointer to the start of the data block being transferred
- u8DataSize is the length, in bytes, of the included data block (this must be less than or equal to the maximum data block length for the client, specified in the Image Block Request)

Parent topic: Structures

tsOTA_WaitForData The following structure contains time information for an Image Block Response. It can be used by a response which reports 'failure', to instruct the client to re-request the data block after a certain waiting time (see Section 49.11.10). It can also be used in 'rate limiting' to specify a new value for the 'block request delay' attribute on the client.

```
typedef struct
{
    uint32 u32CurrentTime;
    uint32 u32RequestTime;
    uint16 u16BlockRequestDelayMs;
}tsOTA_WaitForData;
```

where:

- u32CurrentTime is the current UTC time, in seconds, on the server. If UTC time is not supported by the server, this value should be set to zero
- u32RequestTime is the UTC time, in seconds, at which the client should re-issue an Image Block Request. If the server does not support UTC time (indicated by a zero value for u32CurrentTime), the client should interpret this value as a time delay before re-issuing an Image Block Request

Note: If the client does not support UTC time but both of the above values are non-zero, the client will take the difference between the two times as a time delay before re-issuing an Image Block Request.

- u16BlockRequestDelayMs is used in 'rate limiting' to specify the value of the 'block request delay' attribute for the client - this is minimum time, in milliseconds, that the client must wait between consecutive block requests (the client will update the local attribute with this value). If the server does not support rate limiting or does not need to limit the download rate to the client, this field must be set to 0

Parent topic: Structures

tsOTA_WaitForDataParams The following structure is used in the tsOTA_CallBackMessage structure (see Section 49.11.21) on an OTA Upgrade server. It contains the data needed to notify a client that rate limiting is required or the client must wait to receive an upgrade image.

```
typedef struct
{
   bool_t              bInitialized;
   uint16              u16ClientAddress;
   tsOTA_WaitForData   sWaitForDataPyld;
}tsOTA_WaitForDataParams;
```

where:

- bInitialized is a boolean flag indicating the server's request to the client:
  - TRUE - Implement rate limiting or wait to receive upgrade image
  - FALSE - Otherwise
- u16ClientAddress contains the 16-bit network address of the client
- sWaitForDataPyld is a structure containing the payload for an Image Block Response with status OTA_STATUS_WAIT_FOR_DATA (see Section49.11.15)

Parent topic: Structures

tsOTA_PageReqServerParams The following structure is used in the tsOTA_CallBackMessage structure (see Section 49.11.21) on an OTA Upgrade server. It contains the data from an Image Page Request received from a client.

```
typedef struct
{
    uint8                      u8TransactionNumber;
    bool_t                      bPageReqRespSpacing;
    uint16                     u16DataSent;
    tsOTA_ImagePageRequest      sPageReq;
    tsZCL_ReceiveEventAddress   sReceiveEventAddress;
}tsOTA_PageReqServerParams;
```

where:

- u8TransactionNumber is the Transaction Sequence Number (TSN) which is used in the Image Page Request
- bPageReqRespSpacing is a boolean used to request a spacing between consecutive Image Block Responses:
  - TRUE - Implement spacing
  - FALSE - Otherwise
- u16DataSent indicates the number of data bytes contained in the Image Page Request
- sPageReq is a structure containing the payload data from the Image Page Request (see Section 49.11.9)

- sReceiveEventAddress contains the address of the OTA Upgrade client that made the page request

Parent topic: Structures

tsOTA_PersistedData The following structure contains the persisted data that is stored in Flash memory using the NVM module:

```
typedef struct
{
    tsCLD_AS_Ota sAttributes;
    uint32 u32FunctionPointer;
    uint32 u32RequestBlockRequestTime;
    uint32 u32CurrentFlashOffset;
    uint32 u32TagDataWritten;
    uint32 u32Step;
    uint16 u16ServerShortAddress;
#ifdef OTA_CLD_ATTR_REQUEST_DELAY
    bool_t bWaitForBlockReq;
#endif
    uint8 u8ActiveTag[OTA_TAG_HEADER_SIZE];
    uint8 u8PassiveTag[OTA_TAG_HEADER_SIZE];
    uint8 au8Header[OTA_MAX_HEADER_SIZE];
    uint8 u8Retry;
    uint8 u8RequestTransSeqNo;
    uint8 u8DstEndpoint;
    bool_t bIsCoProcessorImage;
    bool_t bIsSpecificFile;
    bool_t bIsNullImage;
    uint8  u8CoProcessorOTAHeaderIndex;
    uint32 u32CoProcessorImageSize;
    uint32 u32SpecificFileSize;
#ifdef OTA_PAGE_REQUEST_SUPPORT
    tsOTA_PageReqParams sPageReqParams;
#endif
#if (OTA_MAX_CO_PROCESSOR_IMAGES != 0)
    uint8 u8NumOfDownloadableImages;
#endif
#ifdef OTA_INTERNAL_STORAGE
    uint8 u8Buf[4];
#endif
}tsOTA_PersistedData;
```

The fields are for internal use and no knowledge of them is required.

Parent topic: Structures

tsOTA_QuerySpecificFileRequestPayload The following structure contains the payload for a Query Specific File Request which is issued by an OTA Upgrade client to request a device-specific file from the server.

```
typedef struct
{
    uint64 u64RequestNodeAddress;
    uint16 u16ManufacturerCode;
    uint16 u16ImageType;
    uint32 u32FileVersion;
    uint16 u16CurrentZibgeeStackVersion;
}tsOTA_QuerySpecificFileRequestPayload;
```

where:

- u64RequestNodeAddress is the IEEE/MAC address of the node requesting the device-specific file from the server
- u16ManufactuerCode is the ZigBee-assigned manufacturer code of the requesting node (0xFFFF is used to indicate any manufacturer)
- u16ImageType indicates the requested file type - one of the reserved values that are assigned to the device-specific file types (the value should be in the range 0xFFC0 to 0xFFFE, but only 0xFFC0 to 0xFFC2 are currently in use)
- 32FileVersion contains the release and build numbers of the application and stack that correspond to the device-specific file - for details of the format, refer to the *ZigBee Over-the-Air Upgrading Cluster Specification (095264)*
- u16CurrentZ``ig``beeStackVersion contains the version of ZigBee stack that is currently running on the client

Parent topic: Structures

tsOTA_QuerySpecificFileResponsePayload The following structure contains the payload for a Query Specific File Response which is issued by an OTA Upgrade server in response to a request for a device-specific file.

```
typedef struct
{
    uint32 u32FileVersion;
    uint32 u32ImageSize;
    uint16 u16ImageType;
    uint16 u16ManufacturerCode;
    uint8 u8Status;
}tsOTA_QuerySpecificFileResponsePayload;
```

where:

- 32FileVersion contains the release and build numbers of the application and stack that correspond to the device-specific file - this field will take the same value as the equivalent field in the corresponding Query Specific File Request (see Section 49.11.19)
- u32ImageSize is the size of the requested file, in bytes
- u16ImageType indicates the requested file type - this field will take the same value as the equivalent field in the corresponding Query Specific File Request (see Section 49.11.19)
- u16ManufactuerCode is the ZigBee-assigned manufacturer code of the requesting node - this field will take the same value as the equivalent field in the corresponding Query Specific File Request (see Section 49.11.19)
- u8Status indicates whether a suitable file is available for download:
  - OTA_STATUS_SUCCESS: A suitable file is available
  - OTA_STATUS_NO_IMAGE_AVAILABLE: No suitable file is available
    - The other elements of the structure are only included in the case of success.

Parent topic: Structures

tsOTA_CallBackMessage For an OTA event, the eEventType field of the tsZCL_CallBackEvent structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element sClusterCustomMessage, which is itself a structure containing a field pvCustomData. This field is a pointer to the following tsOTA_CallBackMessage structure:

```
typedef struct
{
    teOTA_UpgradeClusterEvents      eEventId;
#ifdef OTA_CLIENT
    tsOTA_PersistedData sPersistedData;
    uint8 au8ReadOTAData[OTA_MAX_BLOCK_SIZE];
    uint8 u8NextFreeImageLocation;
    uint8 u8CurrentActiveImageLocation;
#endif
#ifdef OTA_SERVER
    tsCLD_PR_Ota aServerPrams[OTA_MAX_IMAGES_PER_ENDPOINT+OTA_MAX_CO_
↪PROCESSOR_IMAGES];
    tsOTA_AuthorisationStruct                sAuthStruct;
    uint8  u8ServerImageStartSector;
    bool bIsOTAHeaderCopied;
    uint8 au8ServerOTAHeader[OTA_MAX_HEADER_SIZE+OTA_TAG_HEADER_SIZE];
    tsOTA_WaitForDataParams  sWaitForDataParams;
#ifdef OTA_PAGE_REQUEST_SUPPORT
    tsOTA_PageReqServerParams  sPageReqServerParams;
#endif
#endif
    uint8  u8ImageStartSector[OTA_MAX_IMAGES_PER_ENDPOINT];
    uint8 au8CAPublicKey[22];
    uint8 u8MaxNumberOfSectors;
    union
    {
        tsOTA_ImageNotifyCommand                sImageNotifyPayload;
        tsOTA_QueryImageRequest                 sQueryImagePayload;
        tsOTA_QueryImageResponse                sQueryImageResponsePayload;
        tsOTA_BlockRequest                      sBlockRequestPayload;
        tsOTA_ImagePageRequest                  sImagePageRequestPayload;
        tsOTA_ImageBlockResponsePayload         sImageBlockResponsePayload;
        tsOTA_UpgradeEndRequestPayload          sUpgradeEndRequestPayload;
        tsOTA_UpgradeEndResponsePayload         sUpgradeResponsePayload;
        tsOTA_QuerySpecificFileRequestPayload sQuerySpFileRequestPayload;
        tsOTA_QuerySpecificFileResponsePayload
                sQuerySpFileResponsePayload;
        teZCL_Status                            eQueryNextImgRspErrStatus;
        tsOTA_SignerMacVerify                   sSignerMacVerify;
        tsOTA_ImageVersionVerify                sImageVersionVerify;
        tsOTA_UpgradeDowngradeVerify            sUpgradeDowngradeVerify;
    }uMessage;
}tsOTA_CallBackMessage;
```

where:

- eEventId is the OTA event type (enumerations are detailed in Section49.12.2)
- sPersistedData is the structure (see Section 49.11.18) which contains the persisted data that is stored in Flash memory using the NVM module on the client
- au8ReadOTAData is an array containing the payload data from an Image Block Response
- u8NextFreeImageLocation identifies the next free image location where a new upgrade image can be stored
- u8CurrentActiveImageLocation identifies the location of the currently active image on the client
- aServerPrams is an array containing the server data for each image which can be updated by application
- sAuthStruct is a structure which stores the authorisation state and list of client devices that are authorised for OTA upgrade

- u8ServerImageStartSector identifies the server self-image start-sector
- bIsOTAHeaderCopied specifies whether the new OTA header is copied (TRUE) or not (FALSE)
- au8ServerOTAHeader specifies the current server OTA header
- sWaitForDataParams is a structure containing time information that may need to be modified by the server for inclusion in an Image Block Response (for more information, refer to Section 49.11.15)
- sPageReqServerParams is a structure containing page request information that may need to be modified by the server
- u8ImageStartSector is used to store the image start-sector for each image which is stored or will be stored in the devices internal Flash memory - note that this variable assumes a 32-Kbyte sector size and so, for example, if

  64-Kbyte sectors are used, its value will be twice the actual start-sector value
- au8CAPublicKey specifies the CA public key
- u8MaxNumberOfSectors specifies the maximum number of sectors to be used per image
- uMessage is a union containing the command payload in one of the following forms (depending on the command specified by eEventId):
  - sImageNotifyPayload is a structure containing the payload of an Image Notify command
  - sQueryImagePayload is a structure containing the payload of a Query Next Image Request
  - sQueryImageResponsePayload is a structure containing the payload of a Query Next Image Response
  - sBlockRequestPayload is a structure containing the payload of an Image Block Request
  - sImagePageRequestPayload is a structure containing the payload of an Image Page Request
  - sImageBlockResponsePayload is a structure containing the payload of an Image Block Response
  - sUpgradeEndRequestPayload is a structure containing the payload of an Upgrade End Request
  - sUpgradeResponsePayload is a structure containing the payload of an Upgrade End Response
  - sQuerySpFileRequestPayload is a structure containing the payload of a Query Specific File Request
  - sQuerySpFileResponsePayload is a structure containing the payload of a Query Specific File Response
  - eQueryNextImgRspErrStatus is the status returned from the query image response command handler and can be passed up to the application when there is an error via the callback event E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE_

    ERROR. The returned status value will be either E_ZCL_ERR_INVALID_IMAGE_SIZE or E_ZCL_ERR_INVALID_IMAGE_VERSION
  - sSignerMacVerify is a structure containing the signer's IEEE/MAC address from a new upgrade image and a status field (which is set by the application after verifying the signer's address)
  - sImageVersionVerify is a structure containing the image version received in the query next image response and status field (which is set by the application after verifying the image version)

- sUpgradeDowngradeVerify is a structure containing the image version received in the upgrade end response and a status field (which is set by the application after verifying the image version)

Parent topic: Structures

tsCLD_PR_Ota The following structure contains server parameter data that can be pre-set using the function **eOTA_SetServerParams()** and obtained using **eOTA_GetServerData()**:

```
typedef struct
{
    uint8* pu8Data;
    uint32 u32CurrentTime;
    uint32 u32RequestOrUpgradeTime;
    uint8  u8QueryJitter;
    uint8  u8DataSize;
} tsCLD_PR_Ota;
```

where:

- pu8Data is a pointer to the start of a block of data
- u32CurrentTime is the current UTC time, in seconds, on the server. If UTC time is not supported by the server, this value should be set to zero
- u32RequestOrUpgradeTime is used by the server as the 'request time' and the 'upgrade time' when sending responses to clients:
  - As a 'request time', the value may be included in an Image Block Response (see Section 49.11.10 and Section 49.11.15)
  - As an 'upgrade time', the value will be included in an Upgrade End Response (see Section 49.11.12)
- u8QueryJitteris a value between 1 and 100 (inclusive) which is used by a receiving client to decide whether to reply to an Image Notify message - for information on 'Query Jitter', refer to Section 49.7
- u8DataSize is the length, in bytes, of the data block pointed to by pu8Data

Parent topic: Structures

tsCLD_AS_Ota This structure contains attribute values which are stored as part of the persisted data in Flash memory:

```
typedef struct
{
    uint64 u64UgradeServerID;
    uint32 u32FileOffset;
    uint32 u32CurrentFileVersion;
    uint16 u16CurrentStackVersion;
    uint32 u32DownloadedFileVersion;
    uint16 u16DownloadedStackVersion;
    uint8  u8ImageUpgradeStatus;
    uint16 u16ManfId;
    uint16 u16ImageType;
    uint16 u16MinBlockRequestDelay;
} tsCLD_AS_Ota;
where the structure elements are OTA Upgrade cluster attribute values, as described in [Section 49.3](ota_
↪upgrade_cluster_structure_and_attributes.md#id_accc84e1-e4af-4078-b61f-af69b4b4314c).
```

Parent topic: Structures

tsOTA_ImageVersionVerify The following structure contains the data for an event of the type E_CLD_OTA_INTERNAL_COMMAND_VERIFY_IMAGE_VERSION.

```
typedef struct
{
    uint32 u32NotifiedImageVersion;
    uint32 u32CurrentImageVersion;
    teZCL_Status eImageVersionVerifyStatus;
}tsOTA_ImageVersionVerify;
```

where:

- u32NotifiedImageVersion is the version received in the query next image response
- u32CurrentImageVersion is the version of the running image
- eImageVersionVerifyStatus is a status field which should be updated to E_ZCL_SUCCESS or E_ZCL_FAIL by the application after checking the received image version, to indicate whether the upgrade image has a valid image version

Parent topic: Structures

tsOTA_UpgradeDowngradeVerify The following structure contains the data for an event of the type E_CLD_OTA_INTERNAL_COMMAND_SWITCH_TO_UPGRADE_DOWNGRADE.

```
typedef struct
{
    uint32 u32DownloadImageVersion;
    uint32 u32CurrentImageVersion;
    teZCL_Status eUpgradeDowngradeStatus;
}tsOTA_UpgradeDowngradeVerify;
```

where:

- u32DownloadImageVersion is the version received in upgrade end response
- u32CurrentImageVersion is the version of running image
- eImageVersionVerifyStatus is a status field which should be updated to E_ZCL_SUCCESS or E_ZCL_FAIL by the application after checking the received image version, to indicate whether the upgrade image has a valid image version

Parent topic: Structures

Parent topic:*OTA Upgrade cluster*

Enumerations

teOTA_Cluster The following enumerations represent the OTA Upgrade cluster attributes:

```
typedef enum
{
    E_CLD_OTA_ATTR_UPGRADE_SERVER_ID,
    E_CLD_OTA_ATTR_FILE_OFFSET,
    E_CLD_OTA_ATTR_CURRENT_FILE_VERSION,
    E_CLD_OTA_ATTR_CURRENT_ZIGBEE_STACK_VERSION,
    E_CLD_OTA_ATTR_DOWNLOADED_FILE_VERSION,
    E_CLD_OTA_ATTR_DOWNLOADED_ZIGBEE_STACK_VERSION,
    E_CLD_OTA_ATTR_IMAGE_UPGRADE_STATUS,
    E_CLD_OTA_ATTR_MANF_ID,
```

(continues on next page)

(continued from previous page)

```
    E_CLD_OTA_ATTR_IMAGE_TYPE,
    E_CLD_OTA_ATTR_REQUEST_DELAY
}teOTA_Cluster;
```

The above enumerations are described in the table below.

Enumeration	Attribute
E_CLD_OTA_ATTR_UPGRADE_SERVER_ID	Upgrade Server ID
E_CLD_OTA_ATTR_FILE_OFFSET	File Offset
E_CLD_OTA_ATTR_CURRENT_FILE_VERSION	Current File Version
E_CLD_OTA_ATTR_CURRENT_ZIGBEE_STACK_VERSION	Current ZigBee Stack Version
E_CLD_OTA_ATTR_DOWNLOADED_FILE_VERSION	Downloaded File Version
E_CLD_OTA_ATTR_DOWNLOADED_ZIGBEE_STACK_VERSIO	Downloaded ZigBee Stack Version
E_CLD_OTA_ATTR_IMAGE_UPGRADE_STATUS	Image Upgrade Status
E_CLD_OTA_ATTR_MANF_ID	Manufacturer ID
E_CLD_OTA_ATTR_IMAGE_TYPE	Image Type
E_CLD_OTA_ATTR_REQUEST_DELAY	Minimum Block Request Delay

The above attributes are described in Section 49.3.

Parent topic: Enumerations

teOTA_UpgradeClusterEvents The following enumerations represent the OTA Upgrade cluster events:

```
typedef enum
{
    E_CLD_OTA_COMMAND_IMAGE_NOTIFY,
    E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_REQUEST,
    E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE,
    E_CLD_OTA_COMMAND_BLOCK_REQUEST,
    E_CLD_OTA_COMMAND_PAGE_REQUEST,
    E_CLD_OTA_COMMAND_BLOCK_RESPONSE,
    E_CLD_OTA_COMMAND_UPGRADE_END_REQUEST,
    E_CLD_OTA_COMMAND_UPGRADE_END_RESPONSE,
    E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_REQUEST,
    E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_RESPONSE,
    E_CLD_OTA_INTERNAL_COMMAND_TIMER_EXPIRED,
    E_CLD_OTA_INTERNAL_COMMAND_SAVE_CONTEXT,
    E_CLD_OTA_INTERNAL_COMMAND_OTA_DL_ABORTED,
    E_CLD_OTA_INTERNAL_COMMAND_POLL_REQUIRED,
    E_CLD_OTA_INTERNAL_COMMAND_RESET_TO_UPGRADE,
    E_CLD_OTA_INTERNAL_COMMAND_LOCK_FLASH_MUTEX,
    E_CLD_OTA_INTERNAL_COMMAND_FREE_FLASH_MUTEX,
    E_CLD_OTA_INTERNAL_COMMAND_SEND_UPGRADE_END_RESPONSE,
    E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_BLOCK_RESPONSE,
    E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_DL_ABORT,
    E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_IMAGE_DL_COMPLETE,
    E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_SWITCH_TO_NEW_IMAGE,
    E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_IMAGE_BLOCK_REQUEST,
    E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_BLOCK_RESPONSE,
    E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_ABORT,
    E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_COMPLETE,
    E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_USE_NEW_FILE,
    E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_NO_UPGRADE_END_RESPONSE,
```

(continues on next page)

(continued from previous page)

```
    E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE_ERROR,
    E_CLD_OTA_INTERNAL_COMMAND_VERIFY_SIGNER_ADDRESS,
    E_CLD_OTA_INTERNAL_COMMAND_RCVD_DEFAULT_RESPONSE,
    E_CLD_OTA_INTERNAL_COMMAND_VERIFY_IMAGE_VERSION,
    E_CLD_OTA_INTERNAL_COMMAND_SWITCH_TO_UPGRADE_DOWNGRADE,
    E_CLD_OTA_INTERNAL_COMMAND_REQUEST_QUERY_NEXT_IMAGES,
    E_CLD_OTA_INTERNAL_COMMAND_OTA_START_IMAGE_VERIFICATION_IN_LOW_
↪PRIORITY,
    E_CLD_OTA_INTERNAL_COMMAND_FAILED_VALIDATING_UPGRADE_IMAGE,
    E_CLD_OTA_INTERNAL_COMMAND_FAILED_COPYING_SERIALIZATION_DATA
}teOTA_UpgradeClusterEvents;
```

The above enumerations are described in the table below.

Enumeration	Event Description
E_CLD_OTA_COMMAI	Generated on client when an Image Notify message is received from the server to indicate that a new application image is available for download
E_CLD_OTA_COMMAI	Generated on server when a Query Next Image Request is received from a client to enquire whether a new application image is available for download
E_CLD_OTA_COMMAI	Generated on client when a Query Next Image Response is received from the server (in response to a Query Next Image Request) to indicate whether a new application image is available for download
E_CLD_OTA_COMMAI	Generated on server when an Image Block Request is received from a client to request a block of image data as part of a download
E_CLD_OTA_COMMAI	Generated on server when an Image Page Request is received from a client to request a page of image data as part of a download
E_CLD_OTA_COMMAI	Generated on client when an Image Block Response is received from the server (in response to an Image Block Request) and contains a block of image data which is part of a download
E_CLD_OTA_COMMAI	Generated on server when an Upgrade End Request is received from a client to indicate that the complete image has been downloaded and verified
E_CLD_OTA_COMMAI	Generated on client when an Upgrade End Response is received from the server (in response to an Upgrade End Request) to confirm the end of a download
E_CLD_OTA_COMMAI	Generated on server when a Query Specific File Request is received from a client to request a particular application image
E_CLD_OTA_COMMAI	Generated on client when a Query Specific File Response is received from the server (in response to a Query Specific File Request) to indicate whether the requested application image is available for download
E_CLD_OTA_INTERNA	Generated on client to notify the application that the local one-second timer has expired
E_CLD_OTA_INTERNA	Generated on server or client to prompt the application to store context data in Flash memory
E_CLD_OTA_INTERNA	Generated on a client if the received image is invalid or the client has aborted the image download (allowing the application to request the new image again)
E_CLD_OTA_INTERNA	Generated on client to prompt the application to poll the server for a new application image
E_CLD_OTA_INTERNA	Generated on client to notify the application that the stack is going to reset the device
E_CLD_OTA_INTERNA FLASH_MUTEX	Generated on server or client to prompt the application to lock the mutex used for accesses to external Flash

memory

| |E_CLD_OTA_INTERNAL_COMMAND_FREE_-FLASH_MUTEX|Generated on server or client to prompt the application to unlock the mutex used for accesses to external Flash memory| |E_CLD_OTA_INTERNAL_COMMAND_SEND_UPGRADE_END_RESPONSE|Generated on server to notify the application that the stack is going to send an Upgrade End Response to a client| |E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_BLOCK_RESPONSE|Generated on client to notify the application that Image Block Response has been received for co-processor image| |E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_DL_ABORT|Generated on client to notify the application that download of co-processor image from the server has been aborted| |E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_IMAGE_DL_COMPLETE|Generated on client to notify the application that download of co-processor image from the server has completed| |E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_SWITCH_TO_NEW_IMAGE|Generated on client to notify the application that the upgrade time for a previously downloaded co-processor image has been reached (this event is generated after receiving the Upgrade End Response which contains the upgrade time)| |E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_IMAGE_BLOCK_REQUEST|Generated on server when an Image Block Request is

received from a client to request a block of image data

as part of a download and the server finds that the required image is stored in the co-processor's external storage device

| |E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_BLOCK_RESPONSE|Generated on client when an Image Block Response is received from server as part of a device-specific file download - the event contains a block of file data which the client stores in an appropriate location| |E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_ABORT|Generated on client when the final Image Block Response of a device-specific file download has been received from the server| |E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_COMPLETE|Generated on client following a device-specific file download to indicate that the upgrade time has been reached and the file can now be used by the client| |E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_USE_NEW_FILE|Generated to indicate that a device-specific file download is being aborted and any received data must be discarded by the application| |E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_NO_UPGRADE_END_RESPONSE|Generated to indicate that no Upgrade End Response has been received for a device-specific file download (after three attempts to obtain one)| |E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE_ERROR|This event is generated on the client when a Query Next Image Response message is received from the server, in response to a Query Next Image Request with a status of Invalid Image Size.| |E_CLD_OTA_INTERNAL_COMMAND_VERIFY_SIGNER_ADDRESS|This event is generated to prompt the application to verify the signer address received in a new OTA upgrade image. This event gives control to the application to verify that the new upgrade image came from a trusted source. After checking the signer address, the application should set the status field of the event to E_ZCL_SUCCESS (valid source) or E_ZCL_FAIL (invalid source).| |E_CLD_OTA_INTERNAL_COMMAND_RCVD_DEFAULT_RESPONSE|This event is generated on the client when a default response message is received from the server, in response to a Query Next Image Request, Image Block Request or Upgrade End Request. This is an internal ZCL event that results in an OTA download being aborted, thus activating the callback function for the event E_CLD_OTA_INTERNAL_COMMAND_OTA_DL_ABORTED.| |E_CLD_OTA_INTERNAL_COMMAND_VERIFY_IMAGE_VERSION|This event is generated to prompt the application to verify the image version received in a Query Next Image Response. This event allows the application to verify that the new upgrade image has a valid image version. After checking the image version, the application should set the status field of the event to E_ZCL_SUCCESS (valid version) or E_ZCL_FAIL (invalid version).| |E_CLD_OTA_INTERNAL_COMMAND_SWITCH_TO_UPGRADE_DOWNGRADE|This event is generated to prompt the application to verify the image version received in an upgrade end response. This event allows the application to verify that the new upgrade image has a valid image version.

After checking the image version, the application should set the status field of the event to E_ZCL_SUCCESS (valid version) or E_ZCL_FAIL (invalid version).

| |E_CLD_OTA_INTERNAL_COMMAND_REQUEST_QUERY_NEXT_IMAGES|This event is generated on the client when a co-processor image also requires the client to update its own image. After the first file is downloaded (co-processor image) this event notifies the application to allow it to send a Query Next Image command for its own upgrade image, using the function **eOTA_ClientQueryNextImageRequest()**.| |E_CLD_OTA_INTERNAL_COMMAND_OTA_START_IMAGE_VERIFICATION_IN_LOW_PRIORITY|This event is generated to prompt the application to verify the downloaded JN516x client image from a low priority task. Once the low priority task is running, the application should call **eOTA_VerifyImage()** to start image verification.| |E_CLD_OTA_INTERNAL_COMMAND_FAILED_VALIDATING_UPGRADE_IMAGE|This event is generated on the client when the validation of a new upgrade image fails. This validation takes place when the upgrade time is reached.| |E_CLD_OTA_INTERNAL_COMMAND_FAILED_COPYING_SERIALIZATION_DATA|This event is generated on the client when the copying of serialization data from the active image to the new upgrade image fails. This process takes place after image validation (if applicable) are completed successfully.|

The above events are described in more detail in Section 49.9.

Parent topic: Enumerations

eOTA_AuthorisationState The following enumerations represent the authorisation options concerning which clients are allowed to obtain upgrade images from the server:

```
typedef enum
{
    E_CLD_OTA_STATE_ALLOW_ALL,
    E_CLD_OTA_STATE_USE_LIST
}eOTA_AuthorisationState;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_OTA_STATE_ALLOW	Allow all clients to obtain upgrade images from this server
E_CLD_OTA_STATE_USE_LI	Only allow clients in authorisation list to obtain upgrade images from this server

Parent topic: Enumerations

teOTA_ImageNotifyPayloadType The following enumerations represent the payload options for an Image Notify message issued by the server:

```
typedef enum
{
    E_CLD_OTA_QUERY_JITTER,
    E_CLD_OTA_MANUFACTURER_ID_AND_JITTER,
    E_CLD_OTA_ITYPE_MDID_JITTER,
    E_CLD_OTA_ITYPE_MDID_FVERSION_JITTER
}teOTA_ImageNotifyPayloadType;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_OTA_QUERY_JITTER	Include only 'Query Jitter' in payload
E_CLD_OTA_MANUFACTURER_ID	Include 'Manufacturer Code' and 'Query Jitter' in payload
E_CLD_OTA_ITYPE_MDID_JITTER	Include 'Image Type', 'Manufacturer Code' and 'Query Jitter' in payload
E_CLD_OTA_ITYPE_MDID_FVERSI	Include 'Image Type', 'Manufacturer Code', 'File Version' and 'Query Jitter' in payload

Parent topic: Enumerations

Parent topic:*OTA Upgrade cluster*

Compile-time options To enable the OTA Upgrade cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_OTA
```

In addition, to enable the cluster as a client or server or both, it is also necessary to add one or both of the following to the same file:

```
#define OTA_CLIENT
#define OTA_SERVER
```

Note: The OTA Upgrade cluster must be enabled as a client or server, as appropriate, in the application images to be downloaded using the cluster. The relevant cluster options (see below) should also be enabled for the image.

The following may also be defined in the **zcl_options.h** file.

Optional Attributes (Client only) The OTA Upgrade cluster has attributes on the client side only. The optional attributes may be specified by defining some or all of the following.

Add this line to enable the optional File Offset attribute:

```
#define OTA_CLD_ATTR_FILE_OFFSET
```

Add this line to enable the optional Current File Version attribute:

```
#define OTA_CLD_ATTR_CURRENT_FILE_VERSION
```

Add this line to enable the optional Current ZigBee Stack Version attribute:

```
#define OTA_CLD_ATTR_CURRENT_ZIGBEE_STACK_VERSION
```

Add this line to enable the optional Downloaded File Version attribute:

```
#define OTA_CLD_ATTR_DOWNLOADED_FILE_VERSION
```

Add this line to enable the optional Downloaded ZigBee Stack Version attribute:

```
#define OTA_CLD_ATTR_DOWNLOADED_ZIGBEE_STACK_VERSION
```

Add this line to enable the optional Manufacturer ID attribute:

```
#define        OTA_CLD_MANF_ID
```

Add this line to enable the optional Image Type attribute:

```
#define OTA_CLD_IMAGE_TYPE
```

Add this line to enable the optional Minimum Block Request Delay attribute:

```
#define OTA_CLD_ATTR_REQUEST_DELAY
```

Global Attributes Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_THERMOSTAT_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Number of Images The maximum number of images that can be stored in the Flash memory of the device of a server or client node must be specified as follows, where in this example the maximum is two images:

```
#define OTA_MAX_IMAGES_PER_ENDPOINT            2
```

Note that the active image should not be included.

OTA Block Size The maximum size of a block of image data to be transferred over the air is defined, in bytes, as follows:

```
#define OTA_MAX_BLOCK_SIZE                     100
```

If a large maximum block size is configured, it is recommended to enable fragmentation for data transfers between nodes. Fragmentation is enabled and configured on the sending and receiving nodes as described in the 'Application Design Notes' appendix of the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

Page Requests The 'page request' feature can be enabled on the server and client by adding the line:

```
#define OTA_PAGE_REQUEST_SUPPORT
```

If the page request feature is enabled then the page size (in bytes) and 'response spacing' (in milliseconds) to be inserted into the Image Page Requests can be configured by defining the following macros on the client:

```
#define OTA_PAGE_REQ_PAGE_SIZE                 512
#define OTA_PAGE_REQ_RESPONSE_SPACING          300
```

The above example definitions contain the default values of 512 bytes and 300 ms.

Hardware Versions in OTA Header If hardware versions will be present in the OTA header then in order to enable checks of the hardware versions on the OTA server and client, add:

```
#define OTA_CLD_HARDWARE_VERSIONS_PRESENT
```

Custom Serialization Data To maintain custom serialization data associated with binary images during upgrades on the server or client, add:

```
#define OTA_MAINTAIN_CUSTOM_SERIALISATION_DATA
```

OTA Command Acks To disable APS acknowledgements for OTA commands on the server or client, add:

```
#define OTA_ACKS_ON FALSE
```

If the above define is not included, APS acknowledgements are enabled by default. **They must be enabled for ZigBee certification**, but for increased download speed it may be convenient to disable them during application development. However, they must not be disabled if using fragmentation.

Frequency of Requests (Client only) To avoid flooding the network with continuous packet exchanges, the request messages from the client can be throttled by defining a time interval, in seconds, between consecutive requests. For example, a one-second interval is defined as follows:

```
#define OTA_TIME_INTERVAL_BETWEEN_REQUESTS 1
```

If this time interval is not defined then the time interval, in seconds, between consecutive retries of an unthrottled message request should be defined. For example, a ten-second retry interval is defined as follows:

```
#define OTA_TIME_INTERVAL_BETWEEN_RETRIES 10
```

(valid only if OTA_TIME_INTERVAL_BETWEEN_REQUESTS is not defined)

Upper Limit on Minimum Block Request Delay An upper limit on the value of the Minimum Block Request Delay attribute is defined, in seconds, as follows:

```
#define OTA_BLOCK_REQUEST_DELAY_MAX_VALUE 2
```

In the above example, the limit is set to 2 seconds. If no value is defined, the default value of this limit is 5 seconds.

Device Address Copying On a device whose application image is to be upgraded (client or server), the OTA Upgrade cluster must copy the IEEE/MAC address of the device from the old image to the new image. This copy must be enabled on the device by adding the line:

```
#define OTA_COPY_MAC_ADDRESS
```

No Security Certificate If no security certificate is to be used, it is necessary to remove references to the Certicom security certificate by including the following definition:

```
#define OTA_NO_CERTIFICATE
```

Internal Storage of OTA Upgrade Image on Client An OTA upgrade image can be stored in the devices internal Flash memory on an OTA Upgrade cluster client by including the following definition:

```
#define OTA_INTERNAL_STORAGE
```

In addition, if the OTA upgrade image is encrypted then it needs to be decrypted before being stored in internal Flash memory. This decryption can be enabled by including the following definition:

```
#define INTERNAL_ENCRYPTED
```

Parent topic:*OTA Upgrade cluster*

Build Process Special build requirements must be implemented when building applications that are to participate in OTA upgrades:

1. Certain lines must be included in the makefiles for the applications - see Section 49.14.1
2. The server and client applications must then be built - see Section 49.14.2
3. The (initial) client application must now be prepared and loaded into Flash memory of the client device - see Section 49.14.3
4. The server application must now be prepared and loaded into Flash memory of the server device - see Section 49.14.4

Modifying Makefiles In the makefiles for all applications (for server and all clients), replace the following lines:

```
$(OBJCOPY) -j .version -j .bir -j .flashheader -j .vsr_table
-j .vsr_handlers  -j .rodata -j .text -j .data -j .bss -j .heap
-j .stack -S -O binary $< $@
```

with:

```
        $(OBJCOPY) -j .version -j .bir -j .flashheader -j .vsr_table
-j .vsr_handlers -j .ro_mac_address -j .ro_ota_header  -j .rodata
-j .text -j .data -j .bss -j .heap -j .stack -S -O binary $< $@
```

Parent topic: Build Process

Building Applications The server and client applications must be built with the makefiles adapted for OTA upgrade (see Section 49.14.1). A build can be conducted from MCUXpresso as for any ZigBee PRO application - refer to the *MCUXpresso Installation and User Guide (JNUG3136)*.

The resulting binary files must then be prepared and loaded into Flash memory as described in Section 49.14.3 and Section 49.14.4.

Parent topic: Build Process

Preparing and Downloading Initial Client Image The first time that the client is programmed with an application, the binary image must be loaded into Flash memory on the client device using a Flash programming tool such as the Flash Programmer within MCUXpresso (normally only used in a development environment) or the *DK6 Production Flash Programmer (JN-SW-4407)*.

After this initial image has been loaded, all subsequent client images will be downloaded from the server to the client via the OTA Upgrade cluster.

Parent topic: Build Process

Preparing and Downloading Server Image The server device is programmed by loading a binary image into Flash memory using a Flash programming tool such as the Flash Programmer within MCUXpresso (normally only used in a development environment) or the *DK6 Production Flash Programmer (JN-SW-4407)*.

When a new client image becomes available for the server to distribute, this image must be loaded into the server.

- In a deployed and running system, this image may be supplied via a backhaul network.
- In a development environment, it may be loaded into Flash memory using the Flash Programmer within MCUXpresso.
  - However, this Flash Programmer only allows programming from the start of Flash memory. Therefore, the server application must be re-programmed into the Flash memory as well as the new client image. The server application binary and client application binary must be combined into a single binary image using the DK6 Encryption Tool (JET) before being loaded into the server. This tool is provided in the SDK and is described in the *JET User Guide (JNUG3135)*.

Note: If desired, the initial server image can also include the initial client application. Although there is no need for the server to download this first client application to the client(s), it may be stored in the server in case there is any subsequent need to re-load it into a client.

Parent topic: Build Process

Parent topic:*OTA Upgrade cluster*

OTA Configuration for Internal Flash OTA cluster is enabled through the **ZCL_options.h** file.

The OTA cluster requires initialization of the location where the upgrade image can be stored. The application provides this through ***eOTA_AllocateEndpointOTASpace*** API.

Each page on the device is 512 bytes. The usable flash size is 632 K, with 32 K typically reserved for NVM (start page 1152) and 24 K for customer data. This leaves a usable flash size for image at 576 K.

If we split it into two sections to support OTA. It means 288 K becomes maximum image size. Each 288 K section would be 576 flash pages.

This could be represented as 32 K sectors to keep in line with legacy devices.

So, for allocation to the OTA cluster:

```
uint8 u8MaxSectorPerImage = 0;
uint8 u8StartSector[1] = {9}; /* So next image starts at 9*32*1024 = 288K offset*/
u8MaxSectorPerImage = 9 ; /* 9 *32* 1024 = 288K is the maximum size of the image */
sNvmDefs.u32SectorSize = 512; /* Sector Size = 512 bytes*/
```

The OTA checks for the presence of the well-known Zigbee09 key at a fixed location within the image.

This provides a convenient mechanism to test the decryption of an encrypted image and an additional sanity check to make sure the image is a valid image to progress downloading.

Note: This is not the mechanism for a full image validation. For a better validation of an OTA image, it is recommended that for OTA an encrypted image with the CRC check is used.

Each application note has an OTA_BUILD folder which holds the OTA compatible images.

LinkKey_3.txt is required for creation of the OTA image. It holds the Key which can be used for validation purpose as described above.

The **config OTA_JN518x_Cer_Keys_HA_Light.txt**and **config OTA_JN518x_Cer_Keys_HA_Light_Generic.txt**files provide the OTA image generator with the offset for the key.

Prior to this release the values were **LinkKey_3.txt,02c0,16**, which should now be **LinkKey_3.txt,01b0,16**.

Switching to a new image After an image is programmed into flash and validated as a correct image to switch to, the following steps should then be performed. These are already done as part of the OTA cluster. However, for custom applications not using the OTA cluster, it would be a requirement to switch and run a new image.

```
  /* Offset in bytes into flash for the new image. In this example, new image is at 288K boundary*/
 u32Offset = 0x48000;
  /* Set the active image location */
  psector_SetEscoreImageData(u32Offset, 0);
  /* Disable any interrupts during the switching of the images */
  __disable_irq();
  /* Remap the vector table to point to the ROM instead of the application */
  SYSCON->MEMORYREMAP &= ~(SYSCON_MEMORYREMAP_MAP_MASK << SYSCON_
↪MEMORYREMAP_MAP_SHIFT);
  /* Prevent entering into Low Power Mode when we reset */
  PMC->CTRL &= ~(PMC_CTRL_LPMODE_MASK << PMC_CTRL_LPMODE_SHIFT);
  /* Initiate reset */
  NVIC_SystemReset();
```

Parent topic:OTA Configuration for Internal Flash

Parent topic:*OTA Upgrade cluster*

Part XIII: Appendices This part comprises nine appendices covering topics that include mutex callbacks, attribute reporting, attribute discovery, custom endpoints, manufacturer-specific attributes and commands, the storage of OTA upgrade applications in the devices Internal or External Flash memory, the OTA upgrade of nodes comprising two processors, example code fragments and a glossary of terms.

Appendix A: Mutex callbacks The mutexes are designed such that a call to **ZPS_u8GrabMutexLock()** must be followed by a call to **ZPS_u8ReleaseMutexLock()**. In addition, the call must not be followed by another call to **ZPS_u8GrabMutexLock()**, which means the mutexes are binary rather than counting. This can cause problems if the ZCL takes a mutex via the callback function and then the application wants to lock the mutex to access the shared device structures. Some ZCL clusters also invoke the callback function with E_ZCL_CBET_LOCK_MUTEX multiple times. The counting mutex code below should be used in the application code. When the application wants to access the shared structure, it should call the **vLockZCLMutex()** function (shown in the code extract below), rather than **ZPS_u8GrabMutexLock()**, so that it also participates in the counting mutex rather than directly taking the binary ZPS mutex-protection. Similarly, the shared structure should be released using **vUnlockZCLMutex()**.

The code below uses a single resource for all endpoints and the general callback function. It defines a file scope counter that is the mutex count related to the resource.

At the top of the application source file, create the count and lock/unlock mutex function prototypes (these prototypes may be placed in a header file, if desired):

```
uint32 u32ZCLMutexCount = 0;
void vLockZCLMutex(void);
void vUnlockZCLMutex(void);
```

In both **cbZCL_GeneralCallback()** and **cbZCL_EndpointCallback()**, make the calls:

```
switch(psEvent->eEventType)
 {
 case E_ZCL_CBET_LOCK_MUTEX:
            vLockZCLMutex();
          break;
     case E_ZCL_CBET_UNLOCK_MUTEX:
       vUnlockZCLMutex();
          break;
```

Define the lock/unlock mutex functions and call them from the application when accessing any ZCL shared structure:

```
void vLockZCLMutex(void)
{
        if (u32ZCLMutexCount == 0)
        {
                ZPS_u8GrabMutexLock(mutexZCL,0);
        }
        u32ZCLMutexCount++;
}
void vUnlockZCLMutex(void)
{
        u32ZCLMutexCount--;
        if (u32ZCLMutexCount == 0)
        {
                ZPS_u8ReleaseMutexLock(mutexZCL,0);
        }
}
```

Appendix B: Attribute reporting Attribute reporting involves sending attribute values unsolicited from the cluster server to a client - that is, pushing values from server to client without the client needing to request the values. This mechanism reduces network traffic compared with the client polling the server for attribute values. It also allows a sleeping server to report its attribute values while it is awake.

The server sends an 'attribute report' to the client, where this report can be issued in one of the following ways:

- by a function call in the user application (on the server device)
- automatically by the ZCL (triggered by a change in the attribute value or periodically)

The rules for automatic reporting (see Appendix B.1) can be configured by a remote client by sending a 'configure reporting' command to the server. The same rules apply to 'default reporting' (see Appendix B.2), but are configured locally on the server. The configuration of attribute reporting is described in Appendix B.3. Remote devices can also query the attribute reporting configuration of the server, as described in Appendix B.6.

Sending and receiving attribute reports are described in Appendix B.4 and Appendix B.5.

Attention: Attribute reporting is an optional feature and is not supported by all devices.

Appendix B.1: Automatic attribute reporting Automatic attribute reporting involves two mechanisms:

- A report is triggered by a change in the attribute value of at least a configured minimum amount
- Reports are issued for the attribute periodically at a configured frequency

These mechanisms can operate at the same time. In this case, reports will be issued periodically and additional reports will be issued between periodic reports if triggered by changes in the attribute value.

If reports are triggered by frequent changes in the attribute value, they may add significantly to the network traffic. To manage this traffic, the production of reports for an attribute can be 'throttled'. This involves defining a minimum time-interval between consecutive reports for the attribute. If the attribute value changes within this time-interval since the last report, a new report will not be generated.

Note: If triggered reports are throttled, periodic reports will still be produced as scheduled.

Periodic reporting can be disabled, leaving only triggered reports to be automatically generated. Automatic reporting can also be disabled altogether (both mechanisms). For information on the configuration of automatic reporting, refer to Appendix B.3.

Parent topic:*Appendix B: Attribute reporting*

Appendix B.2: Default reporting For each cluster, the ZCL specification states that certain attributes must be reportable. These attributes are specified in the cluster descriptions in this manual. Reports on these attributes are optional and can be enabled on an individual basis using a 'reportable flag', as described in Appendix B.3.6. The attributes for which the flag is set will always be reported, defining a set of attributes for 'default reporting'.

Default reporting is a form of automatic reporting (see Appendix B.1) for the restricted set of attributes described above. It is configured on the cluster server as described in Appendix B.3.6. The attributes enabled for default reporting are also included in attribute reporting initiated by the server application though a call to the function **eZCL_ReportAllAttributes()**.

Parent topic:*Appendix B: Attribute reporting*

Appendix B.3: Configuring attribute reporting If attribute reporting is to be used by a cluster then the feature must be enabled at compile-time, as detailed in Appendix B.3.1. Then:

- If automatic attribute reporting is to be implemented then the reports must be configured as described in Appendix B.3.5.
- If default reporting is to be implemented then the reports must be configured as described in Appendix B.3.6.

The ZCL configuration for attribute reporting is described in Appendix B.3.7 for users who wish to modify this configuration.

B.3.1: Compile-time Options Attribute reporting is enabled at compile-time by setting the appropriate macros in **zcl_options.h**. The compile-time options relevant to the cluster server and client are listed separately below.

Parent topic: Appendix B.3: Configuring attribute reporting

B.3.2: Server Options

Generate Attribute Reports To enable a server to generate attribute reports according to configured reporting rules, add the following option:

```
#define ZCL_ATTRIBUTE_REPORTING_SERVER_SUPPORTED
```

Note: Attribute reporting does not need to be enabled with this macro if the reports are generated only via function calls.

Handle 'Configure Reporting' Commands To enable a server to handle 'configure reporting' commands and reply with 'configure reporting' responses, add the following option:

```
#define ZCL_CONFIGURE_ATTRIBUTE_REPORTING_SERVER_SUPPORTED
```

Handle 'Read Reporting Configuration' Commands To enable a server to handle 'read reporting configuration' commands and reply with 'read reporting configuration' responses, add the following option:

```
#define ZCL_READ_ATTRIBUTE_REPORTING_CONFIGURATION_SERVER_SUPPORTED
```

Number of Attribute Reports The number of reportable attributes can be set (to n) using the following line:

```
#define ZCL_NUMBER_OF_REPORTS   n
```

The default value is 10.

Number of String Attribute Reports The number of reportable string attributes can be set (to n) using the following line:

```
#define ZCL_NUMBER_OF_STRING_REPORTS   n
```

The default value is 0 (meaning that string attribute reports are disabled by default).

Maximum Size of Reportable String Attribute The maximum size, in bytes, of a string attribute that can be reported can be set (to n) using the following line:

```
#define ZCL_ATTRIBUTE_REPORT_STRING_MAXIMUM_SIZE   n
```

The default value is 32 bytes.

Minimum Attribute Reporting Interval The minimum time-interval, in seconds, between consecutive attribute reports can be set (to n) using the following line:

```
#define ZCL_SYSTEM_MIN_REPORT_INTERVAL   n
```

The default value is 1 second.

Maximum Attribute Reporting Interval The maximum time-interval, in seconds, between consecutive attribute reports can be set (to n) using the following line:

```
#define ZCL_SYSTEM_MAX_REPORT_INTERVAL  n
```

The default value is 61 seconds.

Parent topic: Appendix B.3: Configuring attribute reporting

B.3.3: Client Options

Receive Attribute Reports To enable a client to receive attribute reports from a server, add the following option:

```
#define ZCL_ATTRIBUTE_REPORTING_CLIENT_SUPPORTED
```

Send 'Configure Reporting' Commands To enable a client to send 'configure reporting' commands and handle the 'configure reporting' responses, add the following option:

```
#define ZCL_CONFIGURE_ATTRIBUTE_REPORTING_CLIENT_SUPPORTED
```

Send 'Read Reporting Configuration' Commands To enable a client to send 'read reporting configuration' commands and handle the 'read reporting configuration' responses, add the following option:

```
#define ZCL_READ_ATTRIBUTE_REPORTING_CONFIGURATION_CLIENT_SUPPORTED
```

Parent topic: Appendix B.3: Configuring attribute reporting

B.3.4: General (Server and Client) Options If attribute reporting is to report any attributes of the 'floating point' type, the following macro must also be enabled in **zcl_options.h** on both the server and client:

```
#define ZCL_ENABLE_FLOAT
```

This enables the use of the floating point library to calculate differences in attribute values. If this library is not already used by the application code, enabling it in this way increases the build size of the application by approximately 5 Kbytes.

Parent topic: Appendix B.3: Configuring attribute reporting

B.3.5: Configuring Automatic Attribute Reports (from Client) If automatic attribute reporting is to be employed between a cluster server and client, the reporting rules must be configured. These rules include the following parameters for each attribute:

- Time-interval between consecutive reports in periodic reporting
- Minimum time-interval between consecutive triggered attribute reports
- Minimum change in the attribute value that will trigger an attribute report

Note 1: Setting the periodic reporting time-interval to the special value of 0x0000 disables periodic reporting for the attribute. Setting this time-interval to the special value of 0xFFFF disables automatic reporting completely (periodic and triggered) for the attribute.

Note 2: Automatic attributes reports are normally produced on a timescale of seconds. However, reports generated on the change of an attribute value can be speeded up to occur on a timescale of milliseconds, as described in Appendix B.3.8.

Note 3: Before automatic reporting can be configured on an attribute, the 'reportable flag' must be set for the attribute on the cluster server using the function **eZCL_SetReportableFlag()**.

This configuration is conducted on the cluster server but is normally directed from a remote device via 'configure reporting' commands.

The configuration of automatic attribute reporting follows the process:

1. The client sends a 'configure reporting' command to the server.
2. The server receives and processes the command, configures the attribute reporting and generates a 'configure reporting' response, which it sends back to the requesting client.
3. The client receives the 'configure reporting' response and the ZCL generates events to indicate the status of the request to the client.

These steps are described separately below.

1. Sending a 'Configure Reporting' Command (from Client) The application on the cluster client device can configure attribute reporting for a set of attributes on the cluster server using the function **eZCL_SendConfigureReportingCommand()**. This function sends a 'configure reporting' command to the server.

In this function call, a pointer must be provided to an array of tsZCL_AttributeReportingConfigurationRecord structures, where each structure contains the configuration details for one attribute on which reporting is to be configured (see Section 6.1.5).

2. Receiving a 'Configure Reporting' Command (on Server) The server will automatically process an incoming 'configure reporting' command and perform the required configuration without assistance from the application. For each attribute (in the configuration request), the reporting configuration values are parsed, after which the ZCL generates an event of the type:

E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE

In the tsZCL_CallBackEvent structure (see Section 6.2) for this event:

- The uMessage field contains a structure of the type tsZCL_AttributeReportingConfigurationRecord (see Section 6.1.5).
- The eZCL_Status field indicates the outcome of parsing the configuration values for the attribute (success or failure)

Thus, the configuration of reporting for a set of attributes will result in a sequence of events of the above type, one for each attribute. The application should copy the contents of the tsZCL_AttributeReportingConfigurationRecord structure for each attribute to RAM (for information on storage format, refer to Appendix B.7.2).

Note that the tsZCL_AttributeReportingConfigurationRecord structure for an attribute contains the field u16MaximumReportingInterval which specifies a time-period for periodic reporting. Periodic reporting should not be too frequent, since a sleepy device must wake to send a report and frequent reports are be a significant drain on power resources. Therefore, the period for periodic reporting is not allowed to be set to a value less than sConfig.u16SystemMaximumReportingInterval in the ZCL configuration (see Appendix B.3.7). If a 'configure reporting' command attempts to set a smaller (non-zero) value, the ZCL discards the reporting configuration for this attribute and set the status for this attribute configuration to E_ZCL_CMDS_INVALID_VALUE in the 'configure reporting' response (see below).

Once attribute reporting has been configured (or not) for all the attributes (in the request), a single event is generated of the type:

E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE

Finally, the server generates a 'configure reporting' response and sends it back to the requesting client.

Note: The application and ZCL hold the attribute reporting configuration data in RAM. To preserve this data through episodes of power loss, the application should also save the data to NVM, as described in Appendix B.7.

3. Receiving a 'Configure Reporting' Response (on Client) A 'configure reporting' response from the cluster server contains an Attribute Status Record for each attribute that was included in the corresponding 'configure reporting' command. For each attribute in the response, the ZCL on the client generates an event of the type:

E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE_RESPONSE

In the tsZCL_CallBackEvent structure (see Section 6.2) for this event, the uMessage field contains a structure of the type tsZCL_AttributeReportingConfigurationResponse (see Section 6.1.6). In this structure:

- The eCommandStatus field indicates the status of the attribute reporting configuration for the attribute.
- The tsZCL_AttributeReportingConfigurationRecord structure (Section 6.1.5) contains other data but only the following fields are used:
  - u16AttributeEnum which identifies the attribute
  - u8DirectionIsReceived which should read 0x01 to indicate that reports of the attribute value are received by the client

Once the above event has been generated for each valid attribute in the response, a single E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE_RESPONSE event is generated to conclude the response.

Parent topic: Appendix B.3: Configuring attribute reporting

B.3.6: Configuring Default Reporting (on Server) Default reporting is a form of automatic reporting for a restricted set of attributes (see Appendix B.2). It is configured on the cluster server.

An individual attribute can be configured as potentially reportable through default reporting by setting the 'reportable flag' E_ZCL_AF_RP in either of the following ways:

- The flag can be incorporated in the line for the attribute in the tsZCL_AttributeDefinition structure for the cluster server. For example, in the following line of code, attribute reporting is enabled for the bOnOff attribute of the On/Off cluster:

```
E_CLD_ONOFF_ATTR_ID_ONOFF, (E_ZCL_AF_RD|E_ZCL_AF_SE|**E\_ZCL\_AF\_RP**), E_
↪ZCL_BOOL,
  (uint32)(&((tsCLD_OnOff*)(0))->bOnOff),0},
```

- The flag can be set by the server application by calling the function **eZCL_SetReportableFlag()**.

The reporting of these attributes can be configured by the server application by calling the function **eZCL_CreateLocalReport()** for each attribute. The configuration values are similar to those for automatic reporting, described in Appendix B.3.5. The reporting configuration is passed to the function in a sZCL_AttributeReportingConfigurationRecord structure. The application can then enable default reporting for reportable attributes using the function

vZCL_SetDefaultReporting(), which checks whether the E_ZCL_AF_RP flag has been set for each attribute and, if so, sets the 'default reporting flag' E_ZCL_ACF_RP.

Parent topic: Appendix B.3: Configuring attribute reporting

B.3.7: ZCL Configuration for Attribute Reporting

This section describes aspects of ZCL configuration related to attribute reporting.

Note: The information in this section is only useful to developers who wish to adjust the standard ZCL configuration for attribute reporting.

Each attribute for which automatic reporting is enabled requires a tsZCL_ReportRecord structure. These structures are maintained internally by the ZCL and space for them is allocated on the ZCL heap. The heap is allocated using the **u32ZCL_Heap** macro - for example:

```
PRIVATE uint32 u32ZCL_Heap[
ZCL_HEAP_SIZE(ZCL_NUMBER_OF_ENDPOINTS,
ZCL_NUMBER_OF_TIMERS,
ZCL_NUMBER_OF_REPORTS)];
```

The number of reportable attributes and the maximum/minimum reporting intervals are passed into the internal eZCL_CreateZCL structure via the sConfig parameter - for example:

```
sConfig.u8NumberOfReports = ZCL_NUMBER_OF_REPORTS;
sConfig.u16SystemMinimumReportingInterval =
                        ZCL_SYSTEM_MIN_REPORT_INTERVAL;
sConfig.u16SystemMaximumReportingInterval =
                        ZCL_SYSTEM_MAX_REPORT_INTERVAL;
```

The above macros have default values that can be over-ridden in the application's **zcl_options.h** file, as indicated in Appendix B.3.1.

A server that supports automatic attribute reporting should have the 'reportable flag' E_ZCL_AF_RP set for any attributes that are reportable. While creating a cluster instance, the **vZCL_SetDefaultReporting()** function should be called, which will set the 'default reporting flag' E_ZCL_ACF_RP to enable default reporting for all the attributes that have the E_ZCL_AF_RP flag set. If a server receives a 'configure reporting' command for an attribute that does not have E_ZCL_ACF_RP flag set, it will return an error and not allow the attribute to be reported. This bit setting is also required for attribute reports generated through calls to the function **eZCL_ReportAllAttributes()**.

Attribute definitions will normally have the 'reportable flag' set only for the mandatory reportable attribute. The application on the server can set the E_ZCL_ACF_RP flag for those attributes on which reporting is not mandatory. This can be done using the function **eZCL_SetReportableFlag()**.

Parent topic: Appendix B.3: Configuring attribute reporting

B.3.8: Speeding Up Automatic Attribute Reports

Automatic attribute reports (configured as described in Appendix B.3.5) that are produced on changes in attribute values can be speeded up to occur with millisecond resolution. Normally, these reports can occur on a timescale of seconds, as they are dependent on the E_ZCL_CBET_TIMER (one second) ticks for sampling. However, they can be made to occur on a timescale of milliseconds by providing E_ZCL_CBET_TIMER_MS (one millisecond) ticks.

In order to do this, the following code must be included in the application:

```
sCallBackEvent.eEventType = E_ZCL_CBET_TIMER_MS;
vZCL_EventHandler(&sCallBackEvent);
```

Note that the E_ZCL_CBET_TIMER ticks still need to be generated, as they are used by UTC time and by the ZCL report manager to keep track of time.

Parent topic: Appendix B.3: Configuring attribute reporting

Parent topic:*Appendix B: Attribute reporting*

Appendix B.4: Sending attribute reports If automatic attribute reporting has been configured between the cluster server and a client (as described in Appendix B.3), the reporting of the relevant attributes will begin immediately after configuration. Attribute reports are automatically generated:

- periodically with the configured time-interval between consecutive reports
- when the attribute value changes by at least the configured minimum amount

Automatic reporting normally employs both of the above mechanisms simultaneously but can be configured to operate without periodic reporting, if required.

If a periodic report becomes overdue, the event E_ZCL_CBET_REPORT_TIMEOUT is generated on the server.

The application on the server can also generate an attribute report, when needed, by calling one of the following functions:

- **eZCL_ReportAllAttributes()**, which sends an attribute report for all the reportable attributes
- **eZCL_ReportAttribute()**, which sends an attribute report for an individual reportable attribute

The above functions send an attribute report containing the current attribute value(s) to one or more clients specified in the function call. Only the standard attributes can be reported - this does not include manufacturer-specific attributes. Use of these functions for attribute reporting requires no special configuration on the server (but a recipient client will need attribute reporting to be enabled in its compile-time options).

Note: The event E_ZCL_CBET_REPORT_REQUEST is automatically generated on the server before sending an attribute report, allowing the application to update the attribute values in the shared structure, if required.

CAUTION:

The application must not rely on the above event as a prompt to update the shared structure when an attribute changes its value. The event is only generated when the change in attribute value is large enough for an attribute report to be produced. Smaller changes will not result in the event or a report.

Parent topic:*Appendix B: Attribute reporting*

Appendix B.5: Receiving attribute reports In order to receive and parse attribute reports from the cluster server, a client must have attribute reporting enabled in its compile-time options (see Appendix B.3.1).

When an attribute report is received from the server, events are generated and the ZCL software performs the following steps:

1. For each attribute in the attribute report, the ZCL generates an E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTE message for the endpoint callback function, which may or may not take action on this message.

2. On completion of the parsing of the attribute response, the ZCL generates a single E_ZCL_CBET_REPORT_ATTRIBUTES message for the endpoint callback function, which may or may not take action on this message.

Note that:

- The E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTE event has the same fields as the E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE event. In the uMessage field of the tsZCL_CallBackEvent structure (see Section 6.2) for these events, the same structure is used, which is of the type tsZCL_IndividualAttributesResponse. However, the eAttributeStatus field is not updated for an attribute report (only for a 'read attributes' response).
- The E_ZCL_CBET_REPORT_ATTRIBUTES event has the same fields as the E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE event.

Parent topic:*Appendix B: Attribute reporting*

Appendix B.6: Querying attribute reporting configuration Any authorised device in a ZigBee wireless network can obtain the attribute reporting configuration of a cluster server. Such a query follows the process below:

1. The cluster client sends a 'read reporting configuration' command to the server.
2. The server receives and processes the command, retrieves the required configuration information and generates a 'read reporting configuration' response, which it sends back to requesting client.
3. The client receives the 'read reporting configuration' response and the ZCL generates events to inform the application of the reporting configuration.

These steps are described separately below.

Sending a 'Read Reporting Configuration' Command (from Client) The application on the cluster client device can request the attribute reporting configuration on the server using **eZCL_SendConfigureReportingCommand()**. This function sends a 'read reporting configuration' command to the server.

In this function call, a tsZCL_AttributeReadReportingConfigurationRecord structure must be specified which indicates the required configuration information - this structure includes a pointer to an array of records, one per attribute for which reporting configuration information is needed (see Section 6.1.7).

Receiving a 'Read Reporting Configuration' Command (on Server) The server automatically processes an incoming 'read reporting configuration' command without assistance from the application. Callback events are not generated. However, the server generates a 'read reporting configuration' response and send it back to the requesting client.

Receiving a 'Read Reporting Configuration' Response (on Client) A 'read reporting configuration' response from the cluster server contains an Attribute Reporting Configuration Record for each attribute that was included in the corresponding 'read reporting configuration' command. For each attribute in the response, the ZCL on the client generates an event of the type:

E_ZCL_CBET_REPORT_READ_INDIVIDUAL_ATTRIBUTE_CONFIGURATION_RESPONSE

In the tsZCL_CallBackEvent structure (see Section 6.2) for this event, the uMessage field contains a structure of the type tsZCL_AttributeReportingConfigurationResponse (see Section 6.1.6) - this is the same structure as used in attribute reporting configuration, described in Appendix B.3.5.

In this structure:

- The eCommandStatus field indicates the status of the request.

- The tsZCL_AttributeReportingConfigurationRecord structure (see Section 6.1.5) includes:
  - u16AttributeEnum which identifies the attribute
  - other fields containing the attribute reporting configuration information

Once the above event has been generated for each valid attribute in the response, a single E_ZCL_CBET_REPORT_READ_ATTRIBUTE_CONFIGURATION_RESPONSE event is generated to conclude the response.

Parent topic:*Appendix B: Attribute reporting*

Appendix B.7: Storing an attribute reporting configuration During the configuration of automatic attribute reporting, described in Appendix B.3.5, the application on the server must store attribute reporting configuration data in RAM and, optionally, in Non-Volatile Memory (NVM). The storage of this data is described in the sub-sections below.

Persisting an attribute reporting configuration The attribute reporting configuration data is stored in RAM on the cluster server. To allow the server device to recover from an interruption of service involving a loss of power, this configuration data should also be saved in Non-Volatile Memory (NVM). In this case, the attribute reporting configuration data can be recovered from NVM during a 'cold start' of the device and automatic attribute reporting can resume without further configuration.

The storage of attribute reporting configuration data in NVM should be performed during the updates of this data on the server, described in Appendix B.3.5. When an E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE event is generated for an attribute, the contents of the incorporated structure tsZCL_AttributeReportingConfigurationRecord should be saved to NVM as well as to RAM (for information on storage format, refer to Appendix B.7.2). Data storage in NVM can be performed under application control using the Non-Volatile Memory Manager (NVM), described in the *Connectivity Framework Reference Manual*.

On a 'cold start' of the device, the application must retrieve the Attribute Reporting Configuration Record for each attribute from NVM and update the ZCL with the reporting configuration (this must be done after the ZCL has been initialized). To do this, the NVM can be used to retrieve the configuration record for an attribute and the function **eZCL_CreateLocalReport()** must then be called to register this data with the ZCL. This function must not be called for attributes that have not been configured for automatic attribute reporting (e.g. those for which the maximum reporting interval is set to REPORTING_MAXIMUM_TURNED_OFF).

Note: The maximum reporting interval in NVM must be set to REPORTING_MAXIMUM_TURNED_OFF (0xFFFF) during a factory reset in order to prevent reporting from being enabled for attributes for which reporting was not previously enabled.

Parent topic: Appendix B.7: Storing an attribute reporting configuration

Formatting an attribute reporting configuration record The format in which the server application stores attribute reporting configuration data in RAM and, optionally, in NVM is at the discretion of the application developer.

The most general method is to store this data in an array of structures, in which there is one array element for each attribute for which automatic reporting is implemented (the size of this array should correspond to the value of the compile-time option SE_NUMBER_OF_REPORTS - see Appendix B.3.1). The information stored for each attribute may include the relevant cluster ID and endpoint number, as well as details of the configured change that can result in an attribute report. However, this method of data storage may require significant memory space and may only be necessary for more complex applications.

Alternative storage formats for this data are possible which economize on the memory requirements. These methods are outlined below.

Reduced Data Storage A simple extension of the above general scheme uses application knowledge of the attributes being reported. In this case, certain static information about the reportable attributes is built into the compiled application and only the changeable information about these attributes is saved to an array in RAM (and NVM). In this way, the required memory space to store the attribute reporting configuration data is reduced.

An example of this method with five reportable attributes is given below.

```
#define SE_NUMBER_OF_REPORTS 5
typedef struct
  {
   uint16 u16Min;
   uint16 u16Max;
   tuZCL_AttributeReportable uChangeValue;
  } tsLocalStruct;
static tsLocalStruct asLocalConfigStruct[SE_NUMBER_OF_REPORTS];
typedef struct
 {
  uint16 u16AttEnum;
  teZCL_ZCLAttributeType eAttType;
 }tsLocalDefs;
static const tsLocalDefs asLocalDefs[SE_NUMBER_OF_REPORTS] = {
   {TPRC_MATCH_1,E_ZCL_UINT32},
   {TPRC_MATCH_6,E_ZCL_BMAP48},
   {TPRC_MATCH_7,E_ZCL_GINT56},
   {TPRC_MATCH_5,E_ZCL_UINT56},
   {TPRC_MATCH_3,E_ZCL_BOOL}
  };
```

In the above example:

- The fixed data (attribute identifier and type) is held in an array of tsLocalDefs structures, with one array element per attribute - this array is defined at compile-time and therefore does not need to be updated in RAM or persisted in NVM.
- The attribute reporting configuration data is held in an array of tsLocalStruct structures, with one array element per attribute - only this array needs to be updated in RAM and persisted in NVM, thus saving storage space.

Note that both arrays have SE_NUMBER_OF_REPORTS elements and there is a one-to-one correspondence between the elements of the two arrays - elements with the same number relate to the same attribute.

Minimized Data Storage It may be possible to optimize the format in which the attribute reporting configuration data is saved in order to suit the attributes reported. For example, if there are only two attributes to be reported, then it may be sufficient to store the attribute reporting configuration data in a single structure, like the following:

```
typedef struct
{
    uint16   u16MinimumReportingIntervalForAttA;
    uint16   u16MaximumReportingIntervalForAttA;
    zint32   u32AttAReportableChange;
    uint16   u16MinimumReportingIntervalForAttB;
    uint16   u16MaximumReportingIntervalForAttB;
        // Attribute B is a discrete type (for example, a bitmap), so does not have a reportable change
} tsZCL_PersistedAttributeReportingConfigurationRecord;
```

Parent topic: Appendix B.7: Storing an attribute reporting configuration

Parent topic:*Appendix B: Attribute reporting*

Appendix C: Extended attribute discovery 'Extended' attribute discovery is similar to the normal attribute discovery described in Section 2.3.4 except the accessibility of each attribute is additionally indicated as being 'read', 'write' or 'reportable'. The application coding details and compile-time options are different, and are described below.

Appendix C.1: Compile-time options If required, the extended attribute discovery feature must be explicitly enabled on the cluster server and client at compile-time by respectively including the following defines in the **zcl_options.h** files:

```
#define ZCL_ATTRIBUTE_DISCOVERY_EXTENDED_SERVER_SUPPORTED
#define ZCL_ATTRIBUTE_DISCOVERY_EXTENDED_CLIENT_SUPPORTED
```

Parent topic:*Appendix C: Extended attribute discovery*

Appendix C.2: Application coding The application on a cluster client can initiate an extended attribute discovery on the cluster server by calling the **eZCL_SendDiscoverAttributesExtendedRequest()** function, which sends a 'discover attributes extended' request to the server. This function allows a range of attributes to be searched for, defined by:

- The 'start' attribute in the range (the attribute identifier must be specified).
- The number of attributes in the range.

Initially, the start attribute should be set to the first attribute of the cluster. If the discovery request does not return all the attributes used on the cluster server, the above function should be called again with the start attribute set to the next 'undiscovered' attribute. Multiple function calls may be required to discover all of the attributes used on the server.

On receiving a discover attributes extended request, the server handles the request automatically (provided that extended attribute discovery has been enabled in the compile-time options - see above) and replies with a 'discover attributes extended' response containing the requested information.

The arrival of the response at the client results in the event E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_EXTENDED_RESPONSE for each attribute reported in the response. Therefore, multiple events normally result from a single discover attributes extended request. This event contains details of the reported attribute in a tsZCL_AttributeDiscoveryExtendedResponse structure (see Section 6.1.11).

Following the event for the final attribute reported, the event E_ZCL_CBET_DISCOVER_ATTRIBUTES_EXTENDED_RESPONSE is generated to indicate that all attributes from the discover attributes extended response have been reported.

Parent topic:*Appendix C: Extended attribute discovery*

Appendix D: Custom endpoints A ZigBee device and its associated clusters can be registered on an endpoint using the relevant device registration function, from those listed and described in the *ZigBee Devices User Guide (JNUG3131)*. However, it is also possible to set up a custom endpoint which supports selected clusters (rather than a whole ZigBee device and all of its associated clusters). Custom endpoints are particularly useful when using multiple endpoints on a single node - for example, the first endpoint may support a complete ZigBee device (such as a Light Sensor) while one or more custom endpoints are used to support selected clusters.

Appendix D.1: Devices and Endpoints When using custom endpoints, it is important to note the difference between the following 'devices':

- **Physical device:** This is the physical entity which is the network node

- **Logical device:** This is a software entity which implements a specific set of functionality on the node, e.g. On/Off Switch device

A ZigBee network node may contain multiple endpoints, where one endpoint is used to represent the 'physical device' and other endpoints are used to support 'logical devices'. The following rules apply to cluster instances on endpoints:

- All cluster instances relating to a single 'logical device' must reside on a single endpoint.
- The Basic cluster relates to the 'physical device' rather than a 'logical device' instance. There can be only one Basic cluster server for the entire node, which can be implemented in either of the following ways:
  - A single cluster instance on a dedicated 'physical device' endpoint
  - A separate cluster instance on each 'logical device' endpoint, but each cluster instance must use the same tsZCL_ClusterInstance structure (and the same attribute values)

Parent topic:*Appendix D: Custom endpoints*

Appendix D.2: Cluster Creation Functions For each of the following clusters, a creation function is provided which creates an instance of the cluster on an endpoint:

- Basic: **eCLD_BasicCreateBasic()**
- Power Configuration: **eCLD_PowerConfigurationCreatePowerConfiguration()**
- Device Temperature Configuration: **eCLD_DeviceTemperatureConfigurationCreateDeviceTemperatur**
- Identify: **eCLD_IdentifyCreateIdentify()**
- Groups: **eCLD_GroupsCreateGroups()**
- Scenes: **eCLD_ScenesCreateScenes()**
- On/Off: **eCLD_OnOffCreateOnOff()**
- On/Off Switch Configuration: **eCLD_OOSCCreateOnOffSwitchConfig()**
- Level Control: **eCLD_LevelControlCreateLevelControl()**
- Alarms: **eCLD_AlarmsCreateAlarms()**
- Time: **eCLD_TimeCreateTime()**
- Analogue Input (Basic): **eCLD_AnalogInputBasicCreateAnalogInputBasic()**
- Analogue Output (Basic): **eCLD_AnalogOutputBasicCreateAnalogOutputBasic()**
- Binary Input (Basic): **eCLD_BinaryInputBasicCreateBinaryInputBasic()**
- Binary Output (Basic): **eCLD_BinaryOutputBasicCreateBinaryOutputBasic()**
- Multistate Input (Basic): **eCLD_MultistateInputBasicCreateMultistateInputBasic()**
- Multistate Output (Basic): **eCLD_MultistateOutputBasicCreateMultistateOutputBasic()**
- Poll Control: **eCLD_PollControlCreatePollControl()**
- Power Profile: **eCLD_PPCreatePowerProfile()**
- Diagnostics: **eCLD_DiagnosticsCreateDiagnostics()**
- Illuminance Measurement: **eCLD_IlluminanceMeasurementCreateIlluminanceMeasurement()**
- Illuminance Level Sensing: **eCLD_IlluminanceLevelSensingCreateIlluminanceLevelSensing()**
- Temperature Measurement: **eCLD_TemperatureMeasurementCreateTemperatureMeasurement()**
- Pressure Measurement: **eCLD_PressureMeasurementCreatePressureMeasurement()**
- Flow Measurement: **eCLD_FlowMeasurementCreateFlowMeasurement()**

- Relative Humidity Measurement: **eCLD_RelativeHumidityMeasurementCreateRelativeHumidityMeas**
- Occupancy Sensing: **eCLD_OccupancySensingCreateOccupancySensing()**
- Electrical Measurement: **eCLD_ElectricalMeasurementCreateElectricalMeasurement()**
- Colour Control: **eCLD_ColourControlCreateColourControl()**
- Ballast Configuration: **eCLD_BallastConfigurationCreateBallastConfiguration()**
- Thermostat: **eCLD_ThermostatCreateThermostat()**
- Thermostat User Interface Configuration: **eCLD_ThermostatUIConfigCreateThermostatUIConfig()**
- Door Lock: **eCLD_DoorLockCreateDoorLock()**
- IAS Zone: **eCLD_IASZoneCreateIASZone()**
- IAS Ancillary Control Equipment (ACE): **eCLD_IASACECreateIASACE()**
- IAS Warning Device (WD): **eCLD_IASWDCreateIASWD()**
- Price: **eSE_PriceCreate()**
- Demand-Response and Load Control (DRLC): **eSE_DRLCCreate()**
- Simple Metering: **eSE_SMCreate()**
- Commissioning:**eCLD_CommissioningClusterCreateCommissioning()**
- Touchlink Commissioning: **eCLD_ZllCommissionCreateCommission()**
- Appliance Control: **eCLD_ApplianceControlCreateApplianceControl()**
- Appliance Identification: **eCLD_ApplianceIdentificationCreateApplianceIdentification()**
- Appliance Events and Alerts: **eCLD_ApplianceEventsAndAlertsCreateApplianceEventsAndAlerts()**
- Appliance Statistics: **eCLD_ApplianceStatisticsCreateApplianceStatistics()**
- Over-The-Air (OTA) Upgrade: **eOTA_Create()**

More than one of the above functions can be called for the same endpoint in order to create multiple cluster instances on the endpoint.

Note: No more than one server instance and one client instance of a given cluster can be created on a single endpoint (e.g. one Identify cluster server and one Identify cluster client, but no further Identify cluster instances).

The creation functions for clusters are described in the corresponding chapters of this manual.

Parent topic:*Appendix D: Custom endpoints*

Appendix D.3: Custom Endpoint Set-up In order to set up a custom endpoint (supporting selected clusters), you must do the following in your application code:

1. Create a structure for the custom endpoint containing details of the cluster instances and attributes supported - see *Custom Endpoint Structure* below.
2. Initialise the fields of the tsZCL_EndPointDefinition structure for the endpoint.
3. Call the relevant cluster creation function(s) for the cluster(s) to be supported on the endpoint - see Appendix D.2.
4. Call the ZCL function **eZCL_Register()** for the endpoint.

Custom Endpoint Structure In your application code, to set up a custom endpoint you must create a structure containing details of the cluster instances and attributes to be supported on the endpoint. This structure must include the following:

- A definition of the custom endpoint through a tsZCL_EndPointDefinition structure - for example:
  - tsZCL_EndPointDefinition sEndPoint
- A structure containing a set of tsZCL_ClusterInstance structures for the supported cluster instances - for example:

```
typedef struct
{
    tsZCL_ClusterInstance sBasicServer;
    tsZCL_ClusterInstance sBasicClient;
    tsZCL_ClusterInstance sIdentifyServer;
    tsZCL_ClusterInstance sOnOffCluster;
    tsZCL_ClusterInstance sDoorLockCluster;
} tsHA_AppCustomDeviceClusterInstances
```

- For each cluster instance that is not shared with another endpoint, the following should be specified via the relevant tsZCL_ClusterInstance structure:
- Attribute definitions, if any - for example, the tsCLD_Basic structure for the Basic cluster
- Custom data structures, if any - for example, the tsIdentify_CustomStruct structure for the Identify cluster
- Memory for tables or any other resources, if required by the cluster creation function

Note: If a custom endpoint is to co-exist with a device endpoint, the endpoints can share the structures for the clusters that they have in common. Therefore, it is not necessary to define these cluster structures for the custom endpoint, since they already exist for the device endpoint.

Parent topic:*Appendix D: Custom endpoints*

Appendix E: Manufacturer-specific attributes and commands This appendix describes how a manufacturer can add their own custom attributes and commands to a cluster. Attributes and commands are covered in separate sections.

Appendix E.1: Adding Manufacturer-specific Attributes To add a manufacturer-specific attribute to a cluster:

1. Specify your manufacturer ID code in the Node descriptor. Do this in the ZPS Configuration Editor by clicking on **Node Descriptor** for the relevant node and editing the **Manufacturer Code** field on the **Properties** tab.
2. In the **zcl_options.h** file:

a) Define your manufacturer ID code - for example, for a code of 0x1234, add the line:

```
#define ZCL_MANUFACTURER_CODE 0x1234
```

b) Define the macro that enables the use of manufacturer-specific attributes for the cluster - this macro is cluster-specific but for the Electrical Measurement cluster, the relevant line is:

```
#define CLD_ELECTMEAS_ATTR_MAN_SPEC
```

c) Define an Attribute ID for the new attribute (you must not use a value already used by another attribute) - for example, to add an attribute with an ID of 0x0B00 to the Electrical Measurement cluster, the relevant line is:

```
#define E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC 0x0B00
```

3. Add the new attribute to the cluster structure in the cluster's header file - for example, the code below shows the attribute i16ManufacturerSpecific added to the Electrical Measurement cluster (in **ElectricalMeasurement.h**):

```
typedef struct
{
    zbmap32             u32MeasurementType;
#ifdef CLD_ELECTMEAS_ATTR_AC_FREQUENCY
    zuint16             u16ACFrequency;
#endif
#ifdef CLD_ELECTMEAS_ATTR_RMS_VOLTAGE
    zuint16             u16RMSVoltage;
#endif
#ifdef CLD_ELECTMEAS_ATTR_RMS_CURRENT
    zuint16             u16RMSCurrent;
#endif
#ifdef CLD_ELECTMEAS_ATTR_ACTIVE_POWER
    zint16              i16ActivePower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_REACTIVE_POWER
    zint16              i16ReactivePower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_APPARENT_POWER
    zuint16             u16ApparentPower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_POWER_FACTOR
    zint8               i8PowerFactor;
#endif
**\#ifdef CLD\_ELECTMEAS\_ATTR\_MAN\_SPEC**
**    zint16            i16ManufacturerSpecific;**
**\#endif**
} tsCLD_ElectricalMeasurement;
```

4. Add the new attribute to the source (**.c**) file for the cluster, being careful to add the attribute in the correct sequential position - for example, the following code fragment shows the attribute i16ManufacturerSpecific added to the Electrical Measurement cluster (in **ElectricalMeasurement.c**):

```
const tsZCL_AttributeDefinition
asCLD_ElectricalMeasurementClusterAttributeDefinitions[] = {
/* ZigBee Cluster Library Version */
                :

#ifdef CLD_ELECTMEAS_ATTR_POWER_FACTOR
{E_CLD_ELECTMEAS_ATTR_ID_POWER_FACTOR,
E_ZCL_AF_RD,
E_ZCL_INT8,
(uint16)(&((tsCLD_ElectricalMeasurement*)(0))->i8PowerFactor),
0}, /* Optional */
#endif

/* Manufacturer-specific Read-only Attribute */
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC
{E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC,
```

(continues on next page)

(continued from previous page)

```
(E_ZCL_AF_RD|E_ZCL_AF_MS),
E_ZCL_INT16,
(uint16)(&((tsCLD_ElectricalMeasurement*)(0))->i16ManufacturerSpecific),
0}, /* Optional */
#endif

/* Manufacturer-specific Read/Write Attribute */
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC
{E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC,
(E_ZCL_AF_RD|E_ZCL_AF_WR|E_ZCL_AF_MS),
E_ZCL_INT16,
(uint16)(&((tsCLD_ElectricalMeasurement*)(0))->i16ManufacturerSpecific),
0}, /* Optional */
#endif
```

Within your application code, you can remotely read the value of the new attribute using the following function call:

```
eStatus = eZCL_SendReadAttributesRequest(1, 1, <Add cluster here>, FALSE,
&sSendAddress, &u8SequenceNumber, u8NumAtts, TRUE, HA_MANUFACTURER_CODE,
<Add your attribute to the list here>);
```

Parent topic:*Appendix E: Manufacturer-specific attributes and commands*

Appendix E.2: Adding Manufacturer-specific Commands To add a manufacturer-specific command to a cluster:

1. Ensure that a manufacturer ID code has been specified, as described in Appendix E.1.
2. In the **zcl_options.h** file, define a Command ID for the new command (you must not use a value already used by another command) - for example, to add a command with an ID of 0x20 to the Basic cluster, the relevant line is:

   ```
   #define E_CLD_BASIC_CMD_MANU_SPEC 0x20
   ```

3. In your application code, introduce a handling routine for the new command into the command handler function that is registered when the cluster instance is created - for example, in the case of the Basic cluster, this handler function is **eCLD_BasicCommandHandler()**, which is registered when the function **eCLD_BasicCreateBasic()** is called. Do this as follows:

   a) Define a handler routine specifically for the new command - for example, in the case of the Basic cluster, this function may be **eCLD_BasicHandleManuSpecCommand()** and has the prototype:

   ```
   ```

ZPS_tsAfEvent *pZPSevent, tsZCL_EndPointDefinition *psEndPointDefinition, tsZCL_ClusterInstance *psClusterInstance); “‘

```
b\) Add the Command ID and the above command-specific handler function into the registered command␣
↪handler function - for example, in the case of the Basic cluster, the code for **eCLD\_
↪BasicCommandHandler\(\)** would be modified as shown in the fragment below:

   PUBLIC teZCL_Status eCLD_BasicCommandHandler(
   ZPS_tsAfEvent *pZPSevent,
   tsZCL_EndPointDefinition *psEndPointDefinition,
   tsZCL_ClusterInstance *psClusterInstance)
{...
```

(continues on next page)

(continued from previous page)

```
    // SERVER
switch(u8CommandIdentifier)
{
case(E_CLD_BASIC_CMD_RESET_TO_FACTORY_DEFAULTS):
{
eCLD_BasicHandleResetToFactoryDefaultsCommand(pZPSevent, psEndPointDefinition, psClusterInstance);
break;
}
case(E_CLD_BASIC_CMD_MANU_SPEC):
eCLD_BasicHandleManuSpecCommand(pZPSevent, psEndPointDefinition, psClusterInstance);
break;
default:
{
// unlock
eZCL_ReleaseMutex(psEndPointDefinition);
return(E_ZCL_FAIL);
break;
}
}
...
```

}

```
4. Add a command payload structure for the new command into the source (**.c**) file for the cluster - for␣
↪example:
```

typedef struct { uint8 u8PayloadField1; uint16 u16PayloadField2; uint16 u16PayloadField3; uint32 u32PayloadField4; }tsMS_ManuSpecCommand;

```
5. Add a function for sending the command to a remote node into the header (**.h**) and source (**.c**)␣
↪files for the cluster - for example, in the case of the Basic cluster, the function might be:
```

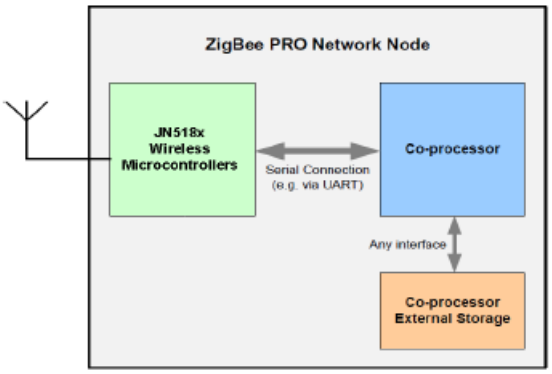
PUBLIC teZCL_Status eCLD_BasicCommandManuSpecSend(uint8 u8SourceEndpoint, uint8 u8DestinationEndpoint, tsZCL_Address *psDestinationAddress, tsMS_ManuSpecCommand *psManuSpecPayload, uint8 *pu8TransactionSequenceNumber) { teZCL_Status eZCL_Status; tsZCL_TxPayloadItem asPayloadDefinition[] = { {1, E_ZCL_UINT8, &psManuSpecPayload->u8PayloadField1}, {1, E_ZCL_UINT16, &psManuSpecPayload->u16PayloadField2}, {1, E_ZCL_UINT16, &psManuSpecPayload->u16PayloadField3}, {1, E_ZCL_UINT32, &psManuSpecPayload->u32PayloadField4} }; eZCL_Status = eZCL_CustomCommandSend(u8SourceEndpoint, u8DestinationEndpoint, psDestinationAddress, GENERAL_CLUSTER_ID_BASIC, TRUE, E_CLD_BASIC_CMD_MANU_SPEC, pu8TransactionSe asPayloadDefinition, TRUE, HA_MANUFACTURER_CODE, sizeof(asPayloadDefinition) / sizeof(tsZCL_TxPayloadItem)); return eZCL_Status; }

```
**Parent topic:**[Appendix E: Manufacturer-specific attributes and commands](../../appendix/topics/
↪manufacturer-specific_attributes_and_commands.md)
```

Appendix F: OTA extension for dual-processor nodes This appendix describes use of the Over-the-Air (OTA) Upgrade cluster (introduced in *Chapter 49*) for a ZigBee PRO network consisting of dual-processor nodes that each contain a JN518x, K32W041, K32W061, MCXW71, or

MCXW72 wireless microcontroller and a co-processor.

The co-processor is connected to the device via a serial interface and may have its own external storage device, as depicted in the figure below.

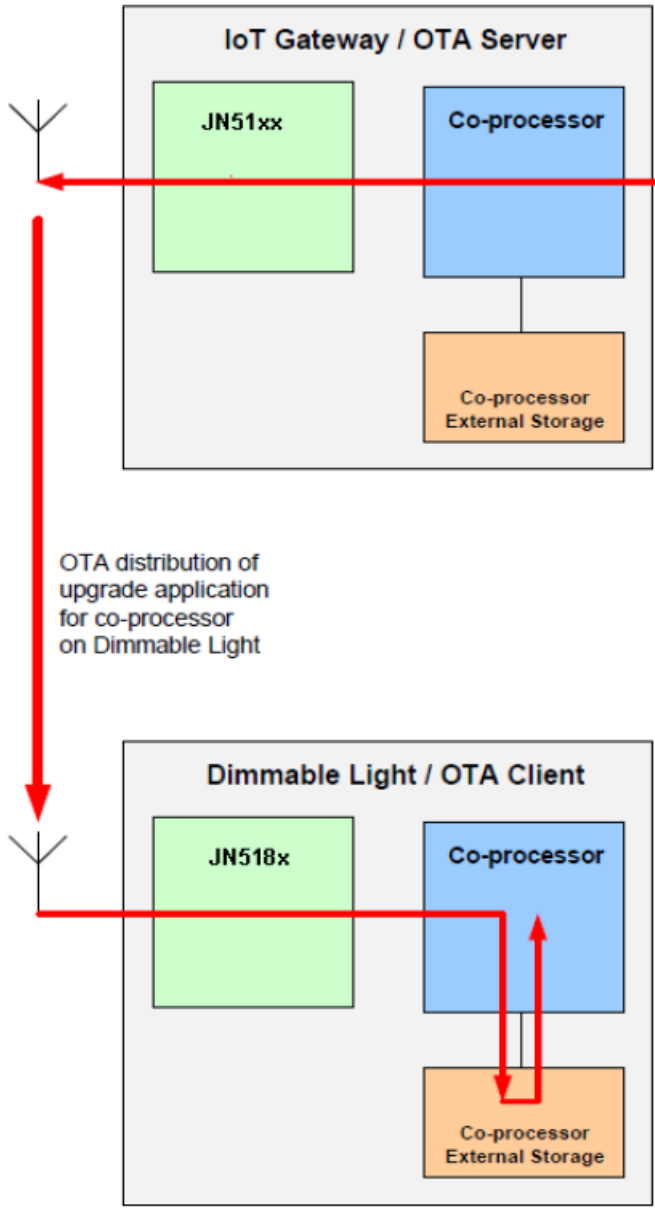


Dual-Processor Node |

The OTA Upgrade cluster may be used to upgrade the application which runs on the co-processor as well as the application which runs on the JN518x, K32W041, K32W061, MCXW71, or MCXW72 device. In this case, the OTA upgrade process is outlined below.

1. On the OTA server node (which is typically also the ZigBee Co-ordinator), the co-processor receives a new software image for the ZigBee PRO network.
2. The co-processor on the OTA server node saves the received software image in its own storage device.
3. The OTA Upgrade cluster server running on the JN518x, K32W041, or K32W061 device distributes the software update over-the-air to the appropriate ZigBee PRO network nodes, as described in Section 49.4.
4. On a target node, the OTA Upgrade cluster client running on the JN518x/K32W041/61 microcontroller either stores the received software image in its own Flash memory device or passes it to the co-processor for storage in the co-processor's own storage device, depending on whether the application in the update is destined for the device or the co-processor.
5. The OTA Upgrade cluster client running on the JN518x, K32W041, or K32W061 device then either performs the upgrade of the application running on itself or signals to the co-processor to initiate an upgrade of its own application, as appropriate.

The above process is illustrated in the figure below for the case of a ZigBee 3.0 network in which the co-processor application on a Dimmable Light (OTA client) is updated from an external source via an 'Internet of Things' (IoT) Gateway (OTA server) and the image is stored in the target co-processor's own storage device.



Example of OTA Upgrade of Co-processor Application |

Appendix F.1: Application Upgrades for Different Target Processors In a ZigBee PRO network containing dual-processor nodes (with a JN518x, K32W041, or K32W061 Wireless microcontroller and a co-processor), an application upgrade can be targeted at any of the following processors:

- OTA server node processors:
  - Wireless Microcontroller (JN518x, K32W041, or K32W061)
  - Co-processor
- OTA client node processors:
  - Wireless Microcontroller (JN518x, K32W041, or K32W061)
  - Co-processor

Only application upgrades for the OTA client node processors need the new software image to be distributed over-the-air.

The following table describes the roles of the different processors (and their associated memory devices) during the different application upgrades.

* If insufficient space in Flash memory, image may be stored in co-processor storage - see Appendix F.2

The case of the co-processor on the OTA server node updating its own application is not described any further in this manual, as this upgrade mechanism is specific to the co-processor. The other three application upgrade scenarios are described in Appendix F.2.

Parent topic:*Appendix F: OTA extension for dual-processor nodes*

Appendix F.2: Storing Upgrade Images in Co-processor Storage on Server When the co-processor on the OTA server node receives a new OTA upgrade image from an external source (such as a utility company), if the image is not for the co-processor itself then it is normally passed to the Wireless Microcontroller device for storage in the attached Flash memory device. However, if there is insufficient storage space in Flash memory then the new image will need to be stored in the storage device of the co-processor:

- When the co-processor application notifies the Wireless Microcontroller application of the arrival of a new image, the Wireless Microcontroller application must check whether there is sufficient Flash memory space for the image.
- If there is insufficient Flash memory space, the Wireless Microcontroller application must inform the co-processor that it should store the image in its own storage device.

The maximum number of images that can be stored in the co-processor's storage device on the OTA server node must be specified as a compile-time option in the **zcl_options.h** file through the macro OTA_MAX_CO_PROCESSOR_IMAGES.

The OTA Upgrade cluster server will require knowledge of any OTA upgrade images stored in the co-processor's storage device - the cluster server must be able to advertise the availability of the image to cluster clients and be able to process requests for the image from clients. To facilitate this role, once the image has been saved, the co-processor must provide the OTA image header information to the Wireless Microcontroller application. The latter application can then register this header information with the cluster server by calling the function **eOTA_NewImageLoaded()**.

When an Image Block Request from a cluster client is received by the cluster server for an image stored in the co-processor's storage device, the event E_CLD_OTA_INTERNAL_COMMAND_CO_PRECOSSOR_IMAGE_BLOCK_REQUEST is generated on the Wireless Microcontroller. After requesting and receiving the required image block from the co-processor, the application must send the block to the relevant client by calling the function **eOTA_ServerImageBlockResponse()** to issue an Image Block Response.

Parent topic:*Appendix F: OTA extension for dual-processor nodes*

Appendix F.3: Use of Image Indices Each OTA upgrade image that is stored in non-volatile memory in a node is identified by an index number. This image index number is actually associated with the memory space allocated to a single image, rather than with a particular image. For example, the image index number 1 may correspond to sectors 3 and 4 of the Flash memory attached to the device.

Note: In the case of external Flash memory, an image index number is linked with the start sector of the memory allocated to a single image when the function **eOTA_AllocateEndpointOTASpace()** is called.

- The maximum number of images that can be stored in the external Flash memory is set at compile-time by defining a value for OTA_MAX_IMAGES_PER_ENDPOINT in the **zcl_options.h** file. The minimum value that can be used is 1, since the active image is held in the internal Flash memory and does not need to be included.

- Since the image indices are numbered from zero, they can take values in the range:
  - 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1)

In the case of a dual-processor node, OTA upgrade images may also be stored in the co-processor's external storage device. The maximum number images that can be stored in this device is set at compile-time by defining a value for OTA_MAX_CO_PROCESSOR_IMAGES in the **zcl_options.h** file.

- The maximum number of images that can be stored across the two storage devices is:
  - OTA_MAX_IMAGES_PER_ENDPOINT + OTA_MAX_CO_PROCESSOR_IMAGES
- The image indices can take values in the range:
  - 0 to (OTA_MAX_IMAGES_PER_ENDPOINT + OTA_MAX_CO_PROCESSOR_IMAGES - 1)
- The indices of the images stored in the external Flash memory still take values in the range:
  - 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1)
- The indices of the images stored in co-processor external storage take values in the range:
  - OTA_MAX_IMAGES_PER_ENDPOINT to (OTA_MAX_IMAGES_PER_ENDPOINT + OTA_MAX_CO_PROCESSOR_IMAGES - 1)

Parent topic:*Appendix F: OTA extension for dual-processor nodes*

Appendix G: Glossary

Term	Description
Address	A numeric value that is used to identify a network node. In ZigBee, the device's
AIB	APS Information Base: A database for the Application Support (APS) layer of th
APDU	Application Protocol Data Unit: Part of a wireless network message that is han
API	Application Programming Interface: A set of programming functions that can l
APS	Application Support: A sub-layer of the Application layer of the ZigBee stack, r
Application	The program that deals with the input/output/processing requirements of the n
Application Profile	A collection of device descriptors that characterise an application for a par-ticu
Attribute	A data entity used by an application, e.g. a temperature measurement. It is par
Binding	The process of associating an endpoint on one node with an endpoint on anoth
Channel	A narrow frequency range within the designated radio band - for example, the
Child	A node which is connected directly to a parent node and for which the par-ent
Cluster	A collection of attributes and commands that define a functional building bloc
Context Data	Data which reflects the current state of the node. The context data must be pre
Co-ordinator	The node through which a network is started, initialised and formed - the Co-o
End Device	A node which has no networking role (such as routing) and is only con-cerned
Endpoint	A software entity that acts as a communications port for an application on a Zi
Extended PAN ID (EPID)	A 64-bit identifier for a ZigBee PRO network that is assigned when the net-work
IEEE 802.15.4	A standard network protocol that is used as the lowest level of the ZigBee softw
IEEE/MAC Address	A unique 64-bit address that is allocated to a device at the time of manufac-tur
Joining	The process by which a device becomes a node of a network. The device trans
Mesh Network	A wireless network topology in which all routing nodes (Routers and the Co-or
Network Address	A 16-bit address that is allocated to a ZigBee node when it joins a network. The
NIB	NWK Information Base: A database containing attributes needed in the manag
Node Descriptor	A set of information about the capabilities of a node.
Node Power Descriptor	A set of information about a node's current and potential power supply.
NPDU	Network Protocol Data Unit: The transmitted form of a wireless network mess
PAN ID	Personal Area Network Identifier: This is a 16-bit value that uniquely identi-fie
Parent	A node which allows other nodes (children) to join the network through it and
Router	A node which provides routing functionality (in addition to input/output/pro-c

Term	Description
Routing	The ability of a node to pass messages from one node to another, acting as a ste
Simple Descriptor	A set of assorted information about a particular application/endpoint.
Sleep Mode	An operating state of a node in which the device consumes minimal power. Du
Stack	The hierarchical set of software layers used to operate a system. The high-level
Stack Profile	The set of features implemented from the ZigBee specification - that is, all the r
UART	Universal Asynchronous Receiver Transmitter: A standard interface used for c
User Descriptor	A user-defined description of a node (e.g. “KitchenLight”).
ZigBee Base Device	A framework for the use of ZigBee device types that provides basic func-tionali
ZigBee Certified Product	An end-product that uses ZigBee Compliant Platforms and public Applica-tion
ZigBee Cluster Library (ZCL)	A collection of clusters that can be individually employed in ZigBee devices, as
ZigBee Compliant Platform	A component (such as a module) that has been tested for ZigBee compli-ance a
ZigBee Device Objects (ZDO)	A special application which resides in the Application Layer on all nodes and p

Revision history The table below lists the revisions to this document.

Document ID	Release date	Description
3.2	11 March 2025	Added support for RW612 devices
3.1	24 January 2025	Added support for MCXW71 and MCXW72 devices
3.0	1 March 2023	- Added support for K32W1 device

- Updated to latest NXP Documentation template

| | |2.0|18 November 2019|Updated for JN518x and K32W041/K32W061| |1.0|20 June 2018|First release|

The following documents provide details on how to demonstrate the capabilities of the NXP Wireless SoC when used as a Zigbee Coprocessor together with a Zigbee application running on the iMX8 (or x86) platform under Linux.

ZigBee NCP Host Coordinator Example

1. General description

List of supported NXP Wireless SoC to act as coprocessor:

- MCXW71-EVK
- FRDM-MCXW71
- K32W061

2. Required Hardware

- 1 x iMX8MMINI-EVK board or 1 x iMX8ULP-EVK board running Linux - Host
- 1 x MCXW71-EVK board or 1 x K32W061 DK6 board or 1 x FRDM-MCXW71 board – Zigbee NCP coprocessor
- 1 x K32W061 DK6 board - Zigbee End Device

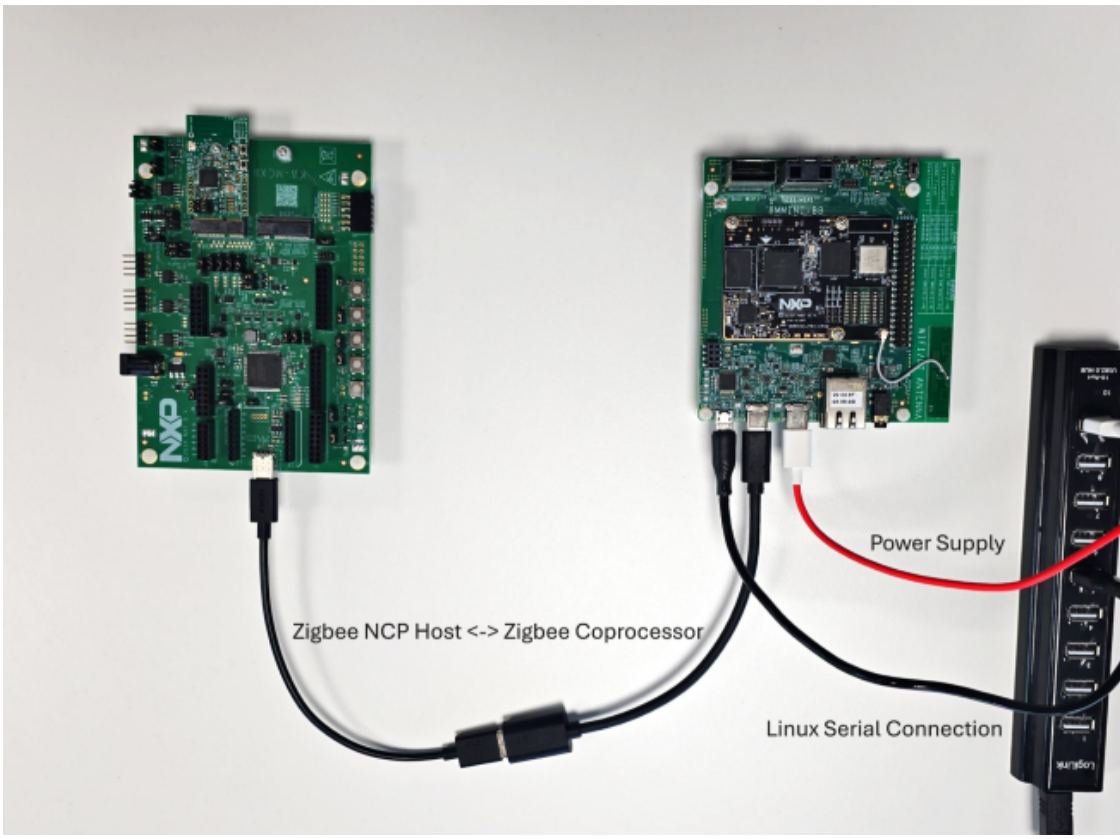
2.1. iMX8 board configuration

2.2. Zigbee NCP coprocessor board configuration

2.2.1 MCXW71-EVK/FRDM-MCXW71 board configuration

2.2.2 K32W061 DK6 board configuration

2.3. K32W061 DK6 board configuration (ZED RX ON)



2.4. Setup Example

3. Building

Environment Setup The NCP Host offers toolchain files that can be used to compile and cross compile the applications. They are available at ZIGBEE_BASE/platform/NCP_HOST/cmake/toolchains and should be provided to cmake -DCMAKE_TOOLCHAIN_FILE command line argument:

- x86_64-linux-gnu.cmake - toolchain file for x86 compile
- arm-linux - toolchain file for imx8 cross-compile

The armgcc toolchain was obtained from official Arm GNU Toolchain website and by default the toolchain file relies on this particular configuration. User can change through environment variables and cmake arguments the default behavior.

- ARMGCC_DIR - path to installed toolchain
- TOOLCHAIN_NAME - toolchain name

The Mbedtls package is required for the encryption/decryption capabilities needed to obtain a secured Serial Link. There are three options to obtain the mbedlts library, options configurable through cmake command line arguments:

- CONFIG_MBEDTLS_SOURCE=SDK - Mbedtls package is obtained from MCUXPRESSO SDK
- CONFIG_MBEDTLS_SOURCE=GIT - Mbedtls package is retrieved from git official repository
- CONFIG_MBEDTLS_SOURCE=SYSTEM - Mbedtls package is used as a preinstalled package

Examples for cross-compile:

- cmake .. -DCMAKE_TOOLCHAIN_FILE=$PWD/../../../../platform/NCP_HOST/cmake/toolchains/arm-linux.cmake -DCONFIG_MBEDTLS_SOURCE=SYSTEM - Cross-compile for imx8 with mbedtls as a preinstalled package. The toolchain file provided detects that machine type is imx8
- cmake .. -DCMAKE_TOOLCHAIN_FILE=$PWD/../../../../platform/NCP_HOST/cmake/toolchains/arm-linux.cmake -DTOOLCHAIN_NAME=aarch64-linux-gnu - Cross-compile for imx8 with toolchain available for Ubuntu 22.04

3.1. iMX8 platform

```
>$ cd out ; cmake .. -DMACHINE_TYPE=imx8
-- The C compiler identification is GNU 13.2.1
-- The CXX compiler identification is GNU GNU 13.2.1
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/aarch64-none-linux-gnu-gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/aarch64-none-linux-gnu-c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Using preinstalled MbedTLS package /usr/local/mbedtls/cmake
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zb-linux-coord/zigbee/examples/zigbee_coordinator/build_linux/
↪out
```

Issue the command make to execute the newly generated Makefile.

```
>[b06830_local@fsr-ub1864-125 out]$ make

[  0%] Building C object CMakeFiles/pdum_static.dir/home/zb-linux-coord/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum.c.o
[  1%] Building C object CCMakeFiles/pdum_static.dir/home/zb-linux-coord/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum_apl.c.o
[  2%] Building C object CMakeFiles/pdum_static.dir/home/zb-linux-coord/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum_dbg.c.o
[  3%] Building C object CMakeFiles/pdum_static.dir/home/zb-linux-coord/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum_nwk.c.o
[  4%] Linking C static library pdum/lib/libpdum.a
[  4%] Built target ncphost-PDUM
[  5%] Building C object CMakeFiles/zb_coord_linux.dir/home/zb-linux-coord/zigbee/examples/zigbee_
↪coordinator/zigbee/examples/zigbee_coordinator/src/linux/pdum_gen_glue.c.o
[  6%] Building C object CMakeFiles/zb_coord_linux.dir/home/zb-linux-coord/zigbee/examples/zigbee_
↪coordinator/zigbee/examples/zigbee_coordinator/src/app_coordinator_ncp.c.o
. . .
[ 99%] Building C object CMakeFiles/zb_coord_linux.dir/home/zb-linux-coord/zigbee/examples/zigbee_
↪coordinator/zigbee/ZCL/Clusters/OTA/Source/OTA.c.o
[100%] Linking C executable zb_coord_linux
[100%] Built target zb_coord_linux
```

3.2. x86 platform

MCUXPRESSO SDK package

```
>$ cd out ; cmake ..
-- The C compiler identification is GNU 11.4.0
-- The CXX compiler identification is GNU 11.4.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found MCUXPRESSO SDK
-- Using SDK root path /home/mcu-sdk
-- Build mbedtls from SDK source code
-- Configuring done
-- Generating done
-- Build files have been written to: /home/mcu-sdk/middleware/wireless/zigbee/examples/zigbee_coordinator/
↪build_linux/out
```

Issue the command `make` to execute the newly generated Makefile.

```
>[b06830_local@fsr-ub1864-125 out]$ make

[  1%] Building C object CMakeFiles/pdum_static.dir/home/mcu-sdk/middleware/wireless/zigbee/platform/
↪NCP_HOST/framework/PDUM/Source/pdum.c.o
[  1%] Building C object CMakeFiles/pdum_static.dir/home/mcu-sdk/middleware/wireless/zigbee/platform/
↪NCP_HOST/framework/PDUM/Source/pdum_apl.c.o
[  2%] Building C object CMakeFiles/pdum_static.dir/home/mcu-sdk/middleware/wireless/zigbee/platform/
↪NCP_HOST/framework/PDUM/Source/pdum_dbg.c.o
[  2%] Building C object CMakeFiles/pdum_static.dir/home/mcu-sdk/middleware/wireless/zigbee/platform/
↪NCP_HOST/framework/PDUM/Source/pdum_nwk.c.o
[  3%] Linking C static library pdum/lib/libpdum.a
[  3%] Built target ncphost-PDUM
[  3%] Building C object mbedtls/library/CMakeFiles/ncp-host-mbedcrypto.dir/aes.c.o
. . .
[ 99%] Building C object CMakeFiles/zb_coord_linux.dir/home/mcu-sdk/middleware/wireless/zigbee//
↪examples/zigbee_coordinator/zigbee/ZCL/Clusters/OTA/Source/OTA.c.o
[100%] Linking C executable zb_coord_linux
[100%] Built target zb_coord_linux
```

The following error might appear, which indicates that the SDK Mbedtls version is too old:

```
-- SDK MBEDTLS version is below 2.28, use CONFIG_MBEDTLS_SOURCE as GIT or SYSTEM instead
CMake Error at /home/mcu-sdk/middleware/wireless/zigbee/platform/NCP_HOST/cmake/
↪NxpZbNcpHostConfig.cmake:46 (message):
Call Stack (most recent call first):
  CMakeLists.txt:32 (include)
```

If so, switch the Mbedtls origin to GIT instead, by building with the following commands:

```
>$ cd out ; cmake -DCONFIG_MBEDTLS_SOURCE=GIT ..
-- The C compiler identification is GNU 11.4.0
-- The CXX compiler identification is GNU 11.4.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
```

(continues on next page)

(continued from previous page)

```
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found MCUXPRESSO SDK internal
-- Using SDK root path /home/mcu-sdk
-- Populate mbedtls repository
Cloning into 'repo'...
HEAD is now at 8b3f26a5ac Merge pull request #868 from ARMmbed/mbedtls-2.28.0rc0-pr
...

-- Found Python3: /usr/bin/python3.10 (found version "3.10.12") found components: Interpreter
-- Performing Test C_COMPILER_SUPPORTS_WFORMAT_SIGNEDNESS
-- Performing Test C_COMPILER_SUPPORTS_WFORMAT_SIGNEDNESS - Success
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
-- Found Threads: TRUE


-- Configuring done
-- Generating done
-- Build files have been written to: /home/zb-linux-coord/zigbee/examples/zigbee_coordinator/build_linux/
↪out
```

Then issue the command `make` as usual to execute the newly generated Makefile.

Standalone Zigbee module Create a directory `out` under the `build_linux` directory and issue the cmake command.

```
>$ cd out ; cmake ..
-- The C compiler identification is GNU 11.4.0
-- The CXX compiler identification is GNU 11.4.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Populate mbedtls repository
Cloning into 'repo'...
HEAD is now at 8b3f26a5ac Merge pull request #868 from ARMmbed/mbedtls-2.28.0rc0-pr
...

-- Found Python3: /usr/bin/python3.10 (found version "3.10.12") found components: Interpreter
-- Performing Test C_COMPILER_SUPPORTS_WFORMAT_SIGNEDNESS
-- Performing Test C_COMPILER_SUPPORTS_WFORMAT_SIGNEDNESS - Success
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
-- Found Threads: TRUE
```

(continues on next page)

(continued from previous page)

```
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zb-linux-coord/zigbee/examples/zigbee_coordinator/build_linux/
↪out
```

Issue the command `make` to execute the newly generated Makefile.

```
>[b06830_local@fsr-ub1864-125 out]$ make

[  0%] Building C object CMakeFiles/pdum_static.dir/home/zb-linux-coord/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum.c.o
[  1%] Building C object CCMakeFiles/pdum_static.dir/home/zb-linux-coord/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum_apl.c.o
[  2%] Building C object CMakeFiles/pdum_static.dir/home/zb-linux-coord/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum_dbg.c.o
[  3%] Building C object CMakeFiles/pdum_static.dir/home/zb-linux-coord/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum_nwk.c.o
[  4%] Linking C static library pdum/lib/libpdum.a
[  4%] Built target ncphost-PDUM
[  5%] Building C object CMakeFiles/zb_coord_linux.dir/home/zb-linux-coord/zigbee/examples/zigbee_
↪coordinator/zigbee/examples/zigbee_coordinator/src/linux/pdum_gen_glue.c.o
[  6%] Building C object CMakeFiles/zb_coord_linux.dir/home/zb-linux-coord/zigbee/examples/zigbee_
↪coordinator/zigbee/examples/zigbee_coordinator/src/app_coordinator_ncp.c.o
. . .
[ 99%] Building C object CMakeFiles/zb_coord_linux.dir/home/zb-linux-coord/zigbee/examples/zigbee_
↪coordinator/zigbee/ZCL/Clusters/OTA/Source/OTA.c.o
[100%] Linking C executable zb_coord_linux
[100%] Built target zb_coord_linux
```

4. Running the application

4.1. Starting the coordinator

Command to display default latency timer for ttyUSB interface: `cat /sys/bus/usb-serial/devices/ttyUSB0/latency_timer`

Command to modify default latency timer for ttyUSB interface: `echo 1 | sudo tee /sys/bus/usb-serial/devices/ttyUSB0/latency_timer`.

Example to start the Zigbee NCP Coordinator:

```
>root@ubuntu:~ ./zb_coord_linux /dev/ttyACM0
[0] Created NCP Host Task with pid 653686
[0] MAIN
[0] ZQ: Initialised a queue: Handle=565ec8c4 Length=30 ItemSize=96
[0] ZQ: Initialised a queue: Handle=565ece50 Length=30 ItemSize=4
[0] ZQ: Initialised a queue: Handle=565ece68 Length=35 ItemSize=4
[0] ZQ: Initialised a queue: Handle=565ece38 Length=3 ItemSize=48
[0] ZQ: Initialised a queue: Handle=565f3700 Length=20 ItemSize=4
[0] ZQ: Initialised a queue: Handle=565f3728 Length=38 ItemSize=4
[0] serial Link initialised
[0] eOTA_NewImageLoaded status = 1
[2] New max process gap 1
[2] Pkt Type 0010 Set New Max Response Time 2
[212] New max process gap 210
[226] Pkt Type 0012 Set New Max Response Time 13
[1178] Recovered Application State 0 On Network 0
>
```

4.2. Forming the network

```
[3259] Form
[3259] APP-EVT: Event 8, NodeState=0
[3267] BDB: Forming Centralized Nwk
[3285] Pkt Type 0024 Set New Max Response Time 18
[3285] APP-EVT: Request Nwk Formation 00
[3805] Nwk formation took 538 MS
[3805] BDB: APP_vGenCallback [0 4]
ZPS_vSetTCLockDownOverride
[3805] APP-BDB: NwkFormation Success
[3805] APP-ZDO: Network started Channel = 12
```

4.3. Steer the network

```
[5879] Steer
[5879] APP-EVT: Event 6, NodeState=10
[5888] APP-BDB: NwkSteering Success
[5888] APP-EVT: Request Nwk Steering 00
[6765] BDB: APP_vGenCallback [0 9]
[6765] APP-ZDO: New Node 0be0 Has Joined
[7024] BDB: APP_vGenCallback [0 1]
[7024] APP-ZDO: Data Indication Status 00 from 0be0 Src Ep 0 Dst Ep 0 Profile 0000 Cluster 0013
[7101] BDB: APP_vGenCallback [0 14]
[7101] APP-ZDO: Discovery Confirm
```

The APP-ZDO Data Indication message signals that an Zigbee End Device has successfully joined the network and is ready to be controlled.

4.4. Find and Bind

```
>[187966] Find
>[187966] APP-EVT: Event 7, NodeState=10
>[187984] APP-EVT: Find and Bind initiate 00
>[188125] - for identify cluster
>[188125]
>CallBackBDB[188125] BDB ZCL Event 8
>[188216] Pkt Type 0052 Set New Max Response Time 23
>[188392] BDB: APP_vGenCallback [0 1]
>[188395] APP-BDB: F&B Simple Desc response From 0be0 Profle 0104 Device 0100 Ep 1 Version 1
>[188399] APP-BDB: Check For Binding Cluster 0006
>[188399] Pkt Type 8074 Took 111
>[188399] Pkt Type 8074 Set New Max Response Time 111
>[188414] APP-BDB: Bind Created for cluster 0006
>[188420] Remote Bind Dst addr 0be0, Ieee Dst Addr b1d48a734e2b758c Ieee Src 00158d00031f14c8
>[188433] Sending a remote bind request Status =0
>[188436] APP-BDB: Bind Created for target EndPt 1
>[188448] APP-ZDO: Data Indication Status 00 from 0be0 Src Ep 0 Dst Ep 0 Profile 0000 Cluster 8004
>[188612]
>
```

4.5. Toggle commands

```
>[239792] Toggle
>[239792] APP-EVT: Event 5, NodeState=10
>[239792] APP-EVT: Send Toggle Cmd
>[239792] ZCL Attribute Report: Cluster 0006 Attribute 0000 Value 1
```

4.6. ZigBee Over-The-Air (OTA) upgrade

```
[19539] Pkt Type 0002 Set New Max Response Time 23
[22830] Pkt Type 0002 Set New Max Response Time 29
[145090] Pkt Type 0002 Set New Max Response Time 33
[421207] Pkt Type 0002 Set New Max Response Time 49
[607037] Nwk status Ind addr 0be0 Status 11
[607037] BDB: APP_vGenCallback [0 13]
[607037] APP-ZDO: Network status Indication 11 addr 0be0
Upgrade End Req received from 0be0 status 0...
[1014587] BDB: APP_vGenCallback [0 9]
[1014587] APP-ZDO: New Node 0be0 Has Joined
[1014753] BDB: APP_vGenCallback [0 1]
[1014753] APP-ZDO: Data Indication Status 00 from 0be0 Src Ep 0 Dst Ep 0 Profile 0000 Cluster 0013
[1014841] BDB: APP_vGenCallback [0 14]
[1014841] APP-ZDO: Discovery Confirm
```

ZigBee NCP Host Router Example

1. General description

List of supported NXP Wireless SoC to act as coprocessor:

- MCXW71-EVK
- FRDM-MCXW71
- K32W061

2. Required Hardware

- 1 x iMX8MMINI-EVK board or 1 x iMX8ULP-EVK board running Linux - Host
- 1 x MCXW71-EVK board or 1 x K32W061 DK6 board or 1 x FRDM-MCXW71 board – Zigbee NCP coprocessor
- 1 x MCXW71-EVK board or 1 x K32W061 DK6 board or 1 x FRDM-MCXW71 board - Zigbee Coordinator

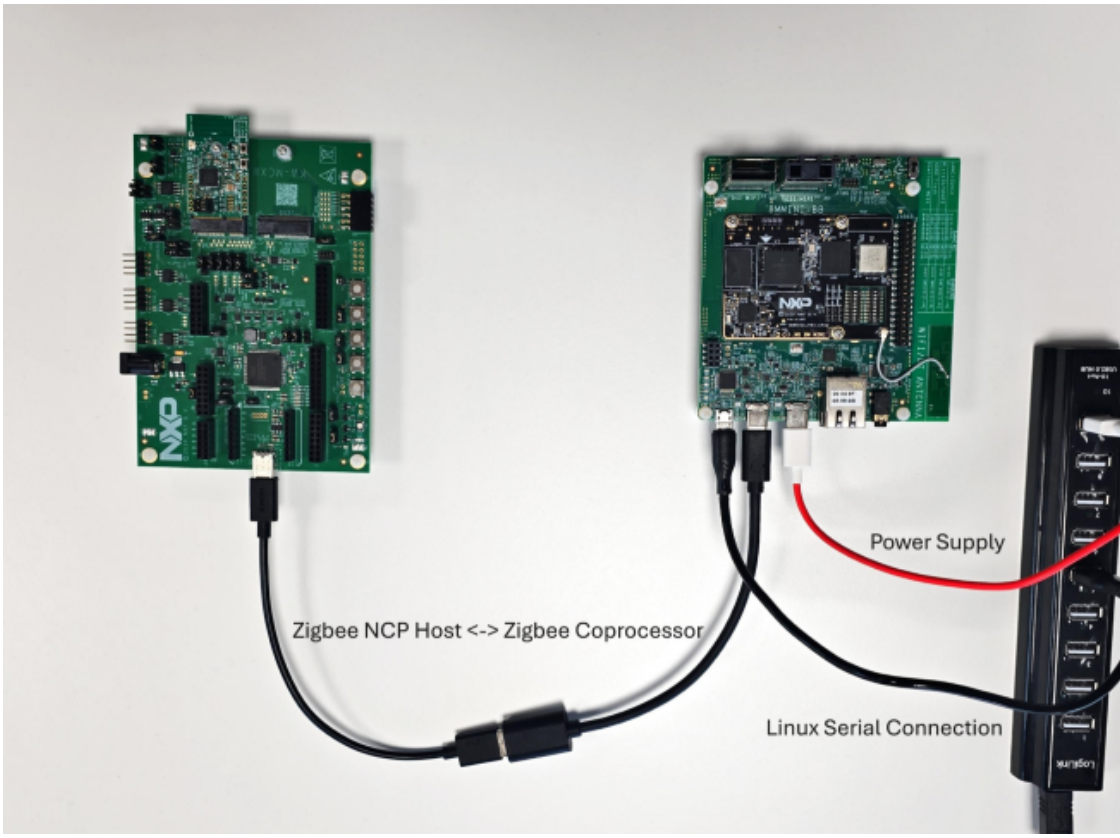
2.1. iMX8 board configuration

2.2. Zigbee NCP coprocessor board configuration

2.2.1 MCXW71-EVK/FRDM-MCXW71 board configuration

2.2.2 K32W061 DK6 board configuration

2.3. K32W061 DK6/ MCXW71-EVK/ FRDM-MCXW71 board configuration (Coordinator)



2.4. Setup Example

3. Building

Environment Setup The NCP Host offers toolchain files that can be used to compile and cross compile the applications. They are available at ZIGBEE_BASE/platform/NCP_HOST/cmake/toolchains and should be provided to cmake -DCMAKE_TOOLCHAIN_FILE command line argument:

- x86_64-linux-gnu.cmake - toolchain file for x86 compile
- arm-linux - toolchain file for imx8 cross-compile

The armgcc toolchain was obtained from official Arm GNU Toolchain website and by default the toolchain file relies on this particular configuration. User can change through environment variables and cmake arguments the default behavior.

- ARMGCC_DIR - path to installed toolchain
- TOOLCHAIN_NAME - toolchain name

The Mbedtls package is required for the encryption/decryption capabilities needed to obtain a secured Serial Link. There are three options to obtain the mbedlts library, options configurable through cmake command line arguments:

- CONFIG_MBEDTLS_SOURCE=SDK - Mbedtls package is obtained from MCUXPRESSO SDK
- CONFIG_MBEDTLS_SOURCE=GIT - Mbedtls package is retrieved from git official repository
- CONFIG_MBEDTLS_SOURCE=SYSTEM - Mbedtls package is used as a preinstalled package

Examples for cross-compile:

- cmake .. -DCMAKE_TOOLCHAIN_FILE=$PWD/../../../../platform/NCP_HOST/cmake/toolchains/arm-linux.cmake -DCONFIG_MBEDTLS_SOURCE=SYSTEM - Cross-compile for imx8 with mbedtls as a preinstalled package. The toolchain file provided detects that machine type is imx8
- cmake .. -DCMAKE_TOOLCHAIN_FILE=$PWD/../../../../platform/NCP_HOST/cmake/toolchains/arm-linux.cmake -DTOOLCHAIN_NAME=aarch64-linux-gnu - Cross-compile for imx8 with toolchain available for Ubuntu 22.04

3.1. iMX8 platform

```
>$ cd out ; cmake .. -DMACHINE_TYPE=imx8
-- The C compiler identification is GNU 13.2.1
-- The CXX compiler identification is GNU GNU 13.2.1
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/aarch64-none-linux-gnu-gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/aarch64-none-linux-gnu-c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Using preinstalled MbedTLS package /usr/local/mbedtls/cmake
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zb-linux-router/zigbee/examples/zigbee_router/build_linux/out
```

Issue the command make to execute the newly generated Makefile.

```
>[b06830_local@fsr-ub1864-125 out]$ make

[  0%] Building C object CMakeFiles/pdum_static.dir/home/zb-linux-router/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum.c.o
[  1%] Building C object CCMakeFiles/pdum_static.dir/home/zb-linux-router/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum_apl.c.o
[  2%] Building C object CMakeFiles/pdum_static.dir/home/zb-linux-router/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum_dbg.c.o
[  3%] Building C object CMakeFiles/pdum_static.dir/home/zb-linux-router/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum_nwk.c.o
[  4%] Linking C static library pdum/lib/libpdum.a
[  4%] Built target ncphost-PDUM
[  5%] Building C object CMakeFiles/zb_router_linux.dir/home/zb-linux-router/zigbee/examples/zigbee_
↪router/zigbee/examples/zigbee_router/src/linux/pdum_gen_glue.c.o
[  6%] Building C object CMakeFiles/zb_router_linux.dir/home/zb-linux-router/zigbee/examples/zigbee_
↪router/zigbee/examples/zigbee_router/src/app_router_node_ncp.c.o
. . .
[ 99%] Building C object CMakeFiles/zb_coord_linux.dir/home/zb-linux-coord/zigbee/examples/zigbee_
↪coordinator/zigbee/ZCL/Clusters/General/Source/OnOffCommands.c.o
[100%] Linking C executable zb_router_linux
[100%] Built target zb_router_linux
```

3.2. x86 platform

MCUXPRESSO SDK package

```
>$ cd out ; cmake ..
-- The C compiler identification is GNU 11.4.0
-- The CXX compiler identification is GNU 11.4.0
```

(continues on next page)

(continued from previous page)

```
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found MCUXPRESSO SDK
-- Using SDK root path /home/mcu-sdk
-- Build mbedtls from SDK source code
-- Configuring done
-- Generating done
-- Build files have been written to: /home/mcu-sdk/middleware/wireless/zigbee/examples/zigbee_router/
↪build_linux/out
```

Issue the command `make` to execute the newly generated Makefile.

```
>[b06830_local@fsr-ub1864-125 out]$ make

[  1%] Building C object CMakeFiles/pdum_static.dir/home/mcu-sdk/middleware/wireless/zigbee/platform/
↪NCP_HOST/framework/PDUM/Source/pdum.c.o
[  1%] Building C object CMakeFiles/pdum_static.dir/home/mcu-sdk/middleware/wireless/zigbee/platform/
↪NCP_HOST/framework/PDUM/Source/pdum_apl.c.o
[  2%] Building C object CMakeFiles/pdum_static.dir/home/mcu-sdk/middleware/wireless/zigbee/platform/
↪NCP_HOST/framework/PDUM/Source/pdum_dbg.c.o
[  2%] Building C object CMakeFiles/pdum_static.dir/home/mcu-sdk/middleware/wireless/zigbee/platform/
↪NCP_HOST/framework/PDUM/Source/pdum_nwk.c.o
[  3%] Linking C static library pdum/lib/libpdum.a
[  3%] Built target ncphost-PDUM
[  3%] Building C object mbedtls/library/CMakeFiles/ncp-host-mbedcrypto.dir/aes.c.o
. . .
[ 99%] Building C object CMakeFiles/zb_router_linux.dir/home/mcu-sdk/middleware/wireless/zigbee//
↪examples/zigbee_router/zigbee/ZCL/Clusters/General/Source/OnOffCommands.c.o
[100%] Linking C executable zb_router_linux
[100%] Built target zb_router_linux
```

The following error might appear, which indicates that the SDK Mbedtls version is too old:

```
-- SDK MBEDTLS version is below 2.28, use CONFIG_MBEDTLS_SOURCE as GIT or SYSTEM instead
CMake Error at /home/mcu-sdk/middleware/wireless/zigbee/platform/NCP_HOST/cmake/
↪NxpZbNcpHostConfig.cmake:46 (message):
Call Stack (most recent call first):
  CMakeLists.txt:32 (include)
```

If so, switch the Mbedtls origin to GIT instead, by building with the following commands:

```
>$ cd out ; cmake -DCONFIG_MBEDTLS_SOURCE=GIT ..
-- The C compiler identification is GNU 11.4.0
-- The CXX compiler identification is GNU 11.4.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
```

(continues on next page)

(continued from previous page)

```
-- Detecting CXX compile features - done
-- Found MCUXPRESSO SDK internal
-- Using SDK root path /home/mcu-sdk
-- Populate mbedtls repository
Cloning into 'repo'...
HEAD is now at 8b3f26a5ac Merge pull request #868 from ARMmbed/mbedtls-2.28.0rc0-pr
...

-- Found Python3: /usr/bin/python3.10 (found version "3.10.12") found components: Interpreter
-- Performing Test C_COMPILER_SUPPORTS_WFORMAT_SIGNEDNESS
-- Performing Test C_COMPILER_SUPPORTS_WFORMAT_SIGNEDNESS - Success
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
-- Found Threads: TRUE


-- Configuring done
-- Generating done
-- Build files have been written to: /home/zb-linux-router/zigbee/examples/zigbee_router/build_linux/out
```

Then issue the command `make` as usual to execute the newly generated Makefile.

Standalone Zigbee module Create a directory `out` under the `build_linux` directory and issue the cmake command.

```
>$ cd out ; cmake ..
-- The C compiler identification is GNU 11.4.0
-- The CXX compiler identification is GNU 11.4.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Populate mbedtls repository
Cloning into 'repo'...
HEAD is now at 8b3f26a5ac Merge pull request #868 from ARMmbed/mbedtls-2.28.0rc0-pr
...


-- Found Python3: /usr/bin/python3.10 (found version "3.10.12") found components: Interpreter
-- Performing Test C_COMPILER_SUPPORTS_WFORMAT_SIGNEDNESS
-- Performing Test C_COMPILER_SUPPORTS_WFORMAT_SIGNEDNESS - Success
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
-- Found Threads: TRUE


-- Configuring done
-- Generating done
-- Build files have been written to: /home/zb-linux-router/zigbee/examples/zigbee_router/build_linux/out
```

Issue the command `make` to execute the newly generated Makefile.

```
>[b06830_local@fsr-ub1864-125 out]$ make

[  0%] Building C object CMakeFiles/pdum_static.dir/home/zb-linux-router/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum.c.o
[  1%] Building C object CCMakeFiles/pdum_static.dir/home/zb-linux-router/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum_apl.c.o
[  2%] Building C object CMakeFiles/pdum_static.dir/home/zb-linux-router/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum_dbg.c.o
[  3%] Building C object CMakeFiles/pdum_static.dir/home/zb-linux-router/zigbee/platform/NCP_HOST/
↪framework/PDUM/Source/pdum_nwk.c.o
[  4%] Linking C static library pdum/lib/libpdum.a
[  4%] Built target ncphost-PDUM
[  5%] Building C object CMakeFiles/zb_coord_linux.dir/home/zb-linux-router/zigbee/examples/zigbee_
↪router/zigbee/examples/zigbee_router/src/linux/pdum_gen_glue.c.o
[  6%] Building C object CMakeFiles/zb_coord_linux.dir/home/zb-linux-router/zigbee/examples/zigbee_
↪router/zigbee/examples/zigbee_router/src/app_router_node_ncp.c.o
. . .
[ 99%] Building C object CMakeFiles/zb_coord_linux.dir/home/zb-linux-coord/zigbee/examples/zigbee_
↪coordinator/zigbee/ZCL/Clusters/General/Source/OnOffCommands.c.o
[100%] Linking C executable zb_router_linux
[100%] Built target zb_router_linux
```

4. Running the application

4.1. Forming the network

On the Zigbee Coordinator, issue the form command on the serial port:

```
eOTA_NewImageLoaded status = 1Recovered Application State 0 On Network 0
form
APP-EVT: Event 8, NodeState=0
APP-EVT: Request Nwk Formation 00
APP-BDB: NwkFormation Success
APP-ZDO: Network started Channel = 12
```

4.2. Steering the network

On the Zigbee Coordinator, issue the steer command on the serial port:

```
steer
APP-EVT: Event 6, NodeState=10
APP-BDB: NwkSteering Success
APP-EVT: Request Nwk Steering 00
```

4.3. Starting the Router

Command to display default latency timer for ttyUSB interface: `cat /sys/bus/usb-serial/devices/ttyUSB0/latency_timer`

Command to modify default latency timer for ttyUSB interface: `echo 1 | sudo tee /sys/bus/usb-serial/devices/ttyUSB0/latency_timer`.

Example to start the Zigbee NCP Router:

```
>root@ubuntu:~ ./zb_router_linux /dev/ttyACM0
[0] Created NCP Host Task with pid 451116
[0] MAIN
[0] ZQ: Initialised a queue: Handle=ad045b60 Length=30 ItemSize=104
[0] ZQ: Initialised a queue: Handle=ad0460c0 Length=30 ItemSize=8
[0] ZQ: Initialised a queue: Handle=ad046100 Length=35 ItemSize=8
```

(continues on next page)

(continued from previous page)

```
[0] ZQ: Initialised a queue: Handle=ad046080 Length=2 ItemSize=64
[0] ZQ: Initialised a queue: Handle=ad04cac0 Length=20 ItemSize=8
[0] ZQ: Initialised a queue: Handle=ad04cb10 Length=38 ItemSize=8
[0] serial Link initialised
[0] eStatusReportReload = 1
[211] New max process gap 1
[214] New max process gap 2
[228] Pkt Type 003a Set New Max Response Time 16
[276] Pkt Type 9094 Set New Max Response Time 17
[819] Pkt Type 0012 Set New Max Response Time 47
[1029] New max process gap 210
[1379] Start Up State 0 On Network 0
>
```

4.4. Joining the network

Once the network is created by the Zigbee Coordinator, the Zigbee NCP Router should join without any additional commands.

```
[5777] BDB: vNsTryNwkJoin - try 0 index 0 of 1 Nwks
[5792] BDB: Try To join 0060377791d75e0f on Ch 21
[5825] APP-ZDO: Discovery Complete 00
[6463] Nwk formation/joining took 6463 MS
...
[7104] Nwk Join Success
[7135] APP: NwkSteering Success
[7135] APP: BDB_EVENT_REJOIN_SUCCESS
[7135] APP-ZDO: Trust Center Status 00
```

4.5. Find and Bind

```
find
APP-EVT: Event 7, NodeState=10
APP-EVT: Find and Bind initiate 00
```

```
[43121] Find
[43121] ZPR: App event 7, NodeState=10
[43129] APP: NwkSteering Success
[43129] APP: BDB_EVENT_REJOIN_SUCCESS
[43593] Update Id 0003
[44617] Update Id 0003
[44617] Update Id 0003
...
```

```
APP-BDB: F&B Simple Desc response From cd7f Profle 0104 Device 0000 Ep 1 Version 0
APP-BDB: Check For Binding Cluster 0006
APP-BDB: Bind Created for cluster 0006
Remote Bind Dst addr cd7f, Ieee Dst Addr 0060377791d75e0f Ieee Src 00158d00031f1224
Sending a remote bind request Status =0
APP-BDB: Bind Created for target EndPt 1
```

```
[45472] BDB: APP_vGenCallback [0 17]
[45472] APP-ZDO: Zdo Bind event
```

4.6. Toggle commands

```
ZCL Attribute Report: Cluster 0006 Attribute 0000 Value 0
ZCL Attribute Report: Cluster 0006 Attribute 0000 Value 0
...
```

```
...
ZCL Attribute Report: Cluster 0006 Attribute 0000 Value 0
toggle
APP-EVT: Event 5, NodeState=10
APP-EVT: Send Toggle Cmd
ZCL Attribute Report: Cluster 0006 Attribute 0000 Value 1
ZCL Attribute Report: Cluster 0006 Attribute 0000 Value 1
```

The following user guides explain how to use Bluetooth LE - Zigbee dynamic mode applications in the context of a Zigbee commissioning over Bluetooth LE use case.

Zigbee coordinator BLE wireless UART demo example This application is provided in the context of demonstrating the Zigbee commissioning over Bluetooth LE and is part of the wireless examples for Zigbee.

Depending on the underlying board, the application is available in two flavours:

- zigbee_coordinator_ble_uart: the zigbee application part is based on the Zigbee Coordinator example and the Bluetooth LE application part is based on the Wireless UART peripheral demo. Targeted boards: MCX-W71-EVK, FRDM-MCXW71, MCX-W72-EVK, FRDM-MCXW72.
- zigbee_coordinator_ble_wu: the zigbee application is based on the Zigbee Coordinator example and the Bluetooth LE application part is based on the EdgeFast Wireless UART demo. Targeted boards: RD-RW612-BGA, FRDM-RW612.

The Narrow Band Unit (NBU) firmware for this application is included in the SDK folder and should be updated based on targeted board. The NBU firmware to be written for Bluetooth LE - Zigbee dynamic mode applications is the same image used for Zigbee example applications.

After reboot, the device running the application is ready to receive commands over Bluetooth LE through the Wireless UART profile. The communication over Bluetooth LE between the application can be intermediated by a mobile application.

To test the application the IoT Toolbox application can be used which is available for both Android and iOS. IoT Toolbox provides an application called ZigBee shell that can be used to send CLI commands over Bluetooth LE through the Wireless UART profile.

The functionality of the Coordinator application is described as follows:

- The Coordinator is responsible for initially forming the network. It also manages other devices that can join the network via the Trust center functionality. It also distributes security materials to those devices that are allowed to join. The Coordinator supports the mandatory clusters and features of the Base Device as defined in the ZigBee Base Device Behavior Specification.
- For demonstrating the "Finding and Binding" functionality, the Coordinator also supports the On/Off Cluster as a client.
- The commands should be sent over Bluetooth LE through the Wireless UART profile. Refer to the *Available CLI commands* chapter for more information on the available commands.

Forming a network A network can be formed from a factory-new Coordinator by sending the zigbee_create_network command. The Coordinator then forms a network. Optionnaly, A ZigBee packet sniffer (running separately on a USB Dongle) might be used to validate if the network is correctly formed. The periodic "link status" messages must be present on the operational channel.

Allowing other devices to join the network Commissioning description and step by step procedure are provided in: commissioning.md

Operating the device The operational functionality of this device in this demonstration is provided by the On/Off cluster. Before being able to send On/Off toggle commands to other devices, the Coordinator must start a Find and Bind as an initiator, this is done by using the zigbee_find command. The other devices must start a Find and Bind procedure as a target. Once bound, the Coordinator can send On/Off toggle commands using zigbee_toggle command to the bound devices.

Rejoining a network As a Coordinator, when this device is restarted in a state that is not factory-new, it resumes operation in its previous state. All applications, bindings, groups, and network parameters are preserved in non-volatile memory.

Performing a factory reset The Coordinator can be returned to its factory-new state, which erases all persistent data except the outgoing network frame counter. To perform a factory reset, send the zigbee_factoryreset command over BLE Wireless UART profile.

LED indication table

LED1	LED2	NOTES
OFF	OFF	The device is not on the network
OFF	ON	The device is active
OFF	Blinking every 1s	Find and Bind active

Available CLI commands

Command	Description
zigbee_factoryrese	Resets the device to its factory state
zigbee_get_netwo	Returns the network status NETWORK_CREATED or NETWORK_NOT_CREATED
zigbee_get_netwo	Returns the network info, Network Info would be serialized in the TLV format if created or NETWORK_NOT_CREATED (if the network is not formed)
zigbee_create_net	Allows to create a zigbee network, NETWORK_ALREADY_CREATED will be retured if the network is already formed
zigbee_find	Start Find & Bind as an initiator, make sure to trigger Find & Bind on a TARGET device
zigbee_toggle	Sends on\off toggle command to bound devices

OTA The Coordinator example supports the OTA Server cluster. As many factors depend on the platform used, please check the platform specific documentation:

- *RW612*

Zigbee Router BLE wireless UART demo example This application is provided in the context of demonstrating the Zigbee commissioning over Bluetooth LE and is part of the wireless examples for Zigbee.

Depending on the underlying board, the application is available in two flavours:

- zigbee_router_ble_uart: the zigbee application part is based on the Zigbee Router example and the Bluetooth LE application part is based on the Wireless UART peripheral demo. Targeted boards: MCX-W71-EVK, FRDM-MCXW71, MCX-W72-EVK, FRDM-MCXW72.

- zigbee_router_ble_wu: the zigbee application is based on the Zigbee Router example and the Bluetooth LE application part is based on the EdgeFast Wireless UART demo. Targeted boards: RD-RW612-BGA, FRDM-RW612.

The Narrow Band Unit (NBU) firmware for this application is included in the SDK folder and should be updated based on targeted board. The NBU firmware to be written for Bluetooth LE - Zigbee dynamic mode applications is the same image used for Zigbee example applications.

After reboot, the device running the application is ready to receive commands over Bluetooth LE through the Wireless UART profile. The communication over Bluetooth LE between the application can be intermediated by a mobile application.

To test the application the IoT Toolbox application can be used which is available for both Android and iOS. IoT Toolbox provides an application called ZigBee shell that can be used to send CLI commands over Bluetooth LE through the Wireless UART profile.

The Zigbee Router demo example demonstrates the Router device type and the "Find and Binding" functionality. To do so, it supports the On/Off Cluster as a server.

A CLI similar to the Coordinator's one is provided. Refer to the *Available CLI commands* chapter for more information on the available commands.

Joining a network The commissioning procedure to join a network is described in: commissioning.md

Operating the device The operational functionality of this device in this demonstration is provided by the On/Off cluster. Since the device supports the On/Off cluster server, its operation is passive, and it responds to commands sent by bound devices. It responds to an OnOff Toggle command from a bound controller device by toggling the LED1 on the board if there's one available.

Rejoining a network As a Router, when this device is restarted in a state, which is not factory-new, it resumes operation in its previous state. All application, binding, group, and network parameters are preserved within the non-volatile memory of the device.

Performing a factory reset The Router can be returned to its factory-new state (erasing all persistent data except the outgoing network frame counter) by sending zigbee_factoryreset command over BLE Wireless UART profile.

The Router then broadcasts a Leave Indication on the old network. It also deletes all persistent data (except the outgoing network frame counter) and performs a software reset. There are two supported over-the-air commands for removing a device from the network. These commands are listed below:

- Network Leave Request without rejoin
- ZDO Management Network Leave Request without rejoin

Binding devices The Router supports the On/Off cluster as a server and implement the "Finding and Binding" process as a target. To trigger "Finding and Binding" as a target, perform the following step:

1. Press the USER button on the board of the target device.
2. Start "Finding and Binding" on the initiator device.

This step causes the Router to self-identify for 180 seconds. In this duration, the initiator tries to find the identifying devices, queries their capabilities, and creates bindings on the devices with matching operational clusters. As part of this process, the Router can receive an Add Group command and/or a Binding Request command.

LED Indicator table

LED1	LED2	Description
OFF	OFF	The device is not on the network
ON/OFF (Current On/Off cluster status) or Blinking (Identifying)	Blinking every 250ms	Find&Bind active

Available CLI commands

Command	Description
zigbee_factoryreset	Resets the device to its factory state
zigbee_get_network_s	Returns the network status CONNECTED or NOT_CONNECTED
zigbee_get_device_inf	Returns the device info (extended address, installation code, CRC), that would be serialized in the TLV format
zigbee_set_network_i	Set the network information, the device will automatically try to join the network at the end of the download

The following documents are provided in PDF format. Later release may convert the pdf documentation to markdown for better review and aligned format.

- MCXW71/MCXW72 and JN518x Core Utilities User Guide pdf
- Zigbee 3.0 Green Power User Guide pdf

For other NXP platforms that are not part of this NXP SDK delivery vehicle namely K32W061/041M/041AM/JN5189/JN5188, in order to find out more about the Zigbee related documentation, please consult the deliverables and the software offerings available at https://www.nxp.com/products/K32W061_41.

Wireless Framework

Wireless Connectivity Framework Connectivity Framework repository provides both connectivity platform enablement with hardware abstraction layer and a set of Services for NXP connectivity stacks : BLE, Zigbee, OpenThread, Matter.

The connectivity framework repository consists of:

- Common folder to common header files for minimal type definition to be used in the repo
- Platform folder used for platform enablement with Hardware abstraction:
  - platform/include: common API header files used by several platforms
  - platform/common: common code for several platforms
  - specifics platform folders , See below the supported platform list
  - platform/../configs folder: configuration files for framework repository and other middlewares (rpmsg, mbedTls, etc.._)

- Services folder
- Zephyr folder for zephyr modules integrated in mcux SDK
- clang formatting script and script folder to format appropriately the source files of the repo

Supported platforms The following devices/platforms are supported in platform folder for connectivity applications:

- kw45x, k32w1x, mcxw71x, under wireless_mcu, kw45_k32w1_mcxw71 folders.
- kw47x, mcxw72x families under wireless_mcu, kw47_mcxw72, kw47_mcxw72_nbu folders.
- rw61x
- RT1060 and RT1170 for Matter
- Other RT devices such as i.MX RT595s

Supported services The supported services are provided for connectivity stacks and their demo application, and are usually dependent on PLATFORM API implementation:

- DBG: Light Debug Module, currently a stubbed header file
- FSCI: Framework Serial Communication Interface between BLE host stack and upper layer located on an other core/device
- FunctionLib: wrapper to toolchain memory manipulation functions (memcpy, memcmp, etc) or use its own implementation for code size reduction
- HWParameters: Store Factory hardware parameters and Application parameters in Flash or IFR
- LowPower: wrapper of SDK power manager for connectivity applications
- ModuleInfo: Store and handle connectivity component versions
- NVM: NXP proprietary File System used for KW45, KW47 automotive devices and RT1060/RT1170 platform for Matter
- OtaSupport: Handle OTA binary writes into internal or external flash.
- SecLib and RNG: Crypto and Random Number generator functions. It supports several ports:
  - Software algorithms
  - Secure subsystem interface to an HW enclave
  - MbedTls 2.x interface
- Sensors: Provides service for Battery and temperature measurements
- SFC: Smart Frequency Calibration to be run from KW47/MCXW71 from NBU core. Matter related modules:
- OTW: Over The Wire module for External Transceiver firmware update from RT platforms
- FactoryDataProvider to be used for Matter

Supported Zephyr modules integration in mcux SDK Connectivity framework provides integration and port layers to the following Zephyr Modules located into zephyr/subsys:

- NVS: Zephyr File System used by Matter and Zigbee

- Settings: Over layer module that allows to store keys into NVS File System used by Matter Port layer and required libraries for these zephyr modules are located in port and lib folder in zephyr directory

Connectivity framework CHANGELOG

7.1.3 mcux SDK 25.12.00 RFP

Major Changes

- [wireless_mcu][wireless_nbu] Replaced interrupt masking macros with static inline functions PLATFORM_SetInterruptMask() and PLATFORM_ClearInterruptMask() to ensure consistent BASEPRI value handling across all compilers. This addresses compiler-dependent behavior issues with the previous macro implementation.
- [wireless_mcu] Added BASEPRI-based interrupt masking in PLATFORM_RemoteActiveRel() to allow high-priority IRQs while ensuring only IMU0 or thread context can call this function.
- [wireless_mcu] Introduced gPlatformUseHwParameter_d compile flag to allow builds without HWParameter section. When undefined or set to 0, crystal trimming functions conditionally access HWParameters only when required.

Minor Changes

- [kw47_mcxw72] Introduced platform-specific library for KW47/MCXW72 platforms.
- [kw47_mcxw72] Support for additional KW47 phantoms including ZB2/ZB3/ZB6/ZB7 and Z83/Z96/Z97.
- [kw47_mcxw72] Removed SH_MEM_TOTAL_SIZE override as it is now automatically calculated to match rpmsg-lite configuration.
- [wireless_mcu] Set RL_BUFFER_PAYLOAD_SIZE to word-aligned value as expected by rpmsg-lite.
- [wireless_mcu] Added system-generated HCI vendor events capability for debug and diagnostic purposes.
- [DBG] Added debug structure transmission over HCI vendor events with NBUDBG_ConfigureHciVendorEvent() API to enable/disable the feature.
- [DBG] Added debug structure versioning and logging buffer address/size information to debug structure.
- [DBG] Removed 15.4 region from debug structure as not supported on KW47.
- [DBG] Added stack overflow detection for armv8_m_mainline architecture.
- [Platform] Simplified enablement of reset features via pin detection
  - Automatically selects gUseResetByLvdForce_c when gAppForceLvdResetOnResetPinDet_d is enabled.
  - Automatically select gUseResetByDeepPowerDown_c when gAppForceDeepPowerDownResetOnResetPinDet_d is enabled.
- [RNG] Replaced gRngHasSecLibDependency_d compilation switch with gRngUseSecLib_d.
- [mcxw23] Defined gRngIsrPrio_c on the preprocessor to make it global and avoid redefinition warnings.
- [mcxw23] Implemented PLATFORM_ResetCrypto() API called by RNG_reinit/SecLib_reinit.

- [mcxw23] Added support of Timer Manager timestamp with OSTIMER.

Bug fixes

- [wireless_mcu] Fixed race condition in PLATFORM_RemoteActiveRel() by adding verification loop to confirm NBU core execution before releasing power domain.
- [wireless_mcu] Added instruction synchronization barrier (__ISB()) after interrupt re-enable in PLATFORM_RemoteActiveRel() to ensure pending interrupts execute between critical sections.
- [wireless_mcu] Fixed external IO voltage isolation issue during low-power initialization - isolation is now cleared at init to ensure proper behavior.
- [wireless_mcu] Replaced spin-wait loops with event-based synchronization in NBU communication APIs. Added mutex protection to PLATFORM_NbuApiReq() and PLATFORM_GetNbuInfo() to prevent race conditions and deadlocks when multiple tasks call these APIs concurrently.
- [wireless_mcu] Fixed OSA bare metal event race condition in ICS where auto-clear event feature could cause tasks to become permanently stuck. Disabled auto-clear feature in bare metal builds and manually clear event flags after OSA_EventWait() returns successfully.
- [NVM] Fixed NvIdle() to prevent looping for more operations than the queue size.
- [NVS] Fixed blank check procedure to return false (non-blank) when checking a 0 length area.
- [NVS] Made external and internal flash ports consistent.
- [DBG] Fixed debug structure size and callback access issues - corrected memory placement overlap between reg_info and assert_info.
- [MISRA] Various MISRA compliance fixes in NVM, HWParameter, LowPower, SecLib, Platform modules and IFR offset definitions.

7.1.2 mcux SDK 25.12.00 pvw2

Major Changes

- [wireless_mcu] Reduced RPMSG buffer payload size from 496 to 270 bytes on KW43/KW47 platforms, saving 226 bytes per buffer (1808 bytes total with 4 buffers on each core). This optimization is possible as rpmsg-lite no longer requires buffer sizes to be powers of two.
- [configs] Introduced RL_ALLOW_CUSTOM_SHMEM_CONFIG flag in rpmsg_config.h to enable connectivity applications to use platform_set_static_shmem_config() and platform_get_custom_shmem_config().

Minor Changes

- [wireless_mcu] Updated radio power management configuration with PLATFORM_InitRadio() API on kw47/mcxw72 platforms
- [DBG] Added NBU assert indication support to host with line/file info using debug structure.
- [DBG] Enhanced NBU debug framework with warning detection and notification capabilities. Extended NBUDBG_StateCheck() to monitor NBU warnings via PLATFORM_IsNbuWarningSet() with callback support for proactive warning monitoring.
- [Sensors] Added periodic temperature measurement support allowing app/host to request periodic temperature measurement.

- [Sensors] Added markdown documentation explaining periodic measurement functionality upon NBU requests.
- [SecLib_RNG] Added documentation for asynchronous seed handling using RNG_NotifyReseedNeeded() and SecLib implementation flavors (Software, EdgeLock, PSA Crypto, MbedTLS).
- [PSA] Simplified PSA configuration files and reduced imports/definitions for wireless MCU platforms.
- [mcxw23] Changed temperature dummy value for sensors to progress on application enablement.
- [NVS] Enhanced debug capabilities by adding CONFIG_NVS_LOG_LEVEL and improved LOG macros to adapt to PRINTF limitations.

Bug fixes

- [wireless_mcu] Fixed FRO32K as 32 kHz clock source with deferred OSC32K switching to improve initialization performance after warm reset.
- [wireless_mcu] Added wait loop for NBU power domain readiness in PLATFORM_InitNbu() to prevent race conditions when accessing NBU power domain in applications without NBU images.
- [wireless_mcu] Fixed external flash blank check procedure for LSPI external NOR Flash by correcting PLATFORM_ExternalFlashAreaIsBlank() to read from RAM buffer and perform erase pattern comparison in RAM with optimized 4-byte step loops.
- [NVS] Removed mflash dependency from NVS external flash port and fixed internal flash blank check of unaligned flash data.
- [mcxw23] Ensured TX power does not exceed 6dBm maximum limit.

7.1.1 mcux SDK 25.12.00 pvw1

Minor Changes

- [wireless_mcu][wireless_nbu] Migrated TSTMR implementation to use SDK fsl_tstmr driver for better maintainability and consistency. This migration replaces custom TSTMR register definitions with official SDK driver APIs while maintaining existing PLATFORM_* API compatibility.

Bug fixes

- [SecLib_RNG][mbedtls] Enhanced ECDH context preservation across low-power transitions on KW45_MCXW71 and KW47_MCXW72 platforms using export/import APIs to ensure cryptographic context is retained when hardware accelerator loses internal memory during power-down mode.
- [wireless_nbu] Fixed incorrect FRODIV values that were causing reduced peripheral clocks by updating PLATFORM_FroDiv[] mapping array to prevent over-division of flash APB and RF_CMC clocks.
- [rw61x/rt1060/rt1170] Added missing common files for external and OTA support on rw61x/rt1060/rt1170 platforms.

7.0.3 RFP mcux SDK 25.09.00

Major Changes

- [wireless_mcu] Replaced ICS RX linked list with message queue to eliminate memory allocation in ISR context and enable user callbacks to run in thread context where memory allocation is permitted.
- [wireless_mcu] Added HCI RX workqueue processing support to reduce ISR execution time and system impact. Feature controlled by gPlatformHciUseWorkqueueRxProcessing_d configuration option (enabled by default on kw45_k32w1_mcxw71 and kw47_mcxw72 platforms in freertos applications). When enabled, HCI transport processes received data in system workqueue thread, allowing user callbacks to run in thread context.
- [wireless_nbu] Introduced PLATFORM_ConfigureSmuDmemMapping() API to configure SMU and DMEM sharing on NBU for kw47_mcxw72 platform, using linker file symbols for correct configuration.
- [mcxw23] Implemented HCI interface using PLATFORM API as preliminary requirement for Zephyr enablement, introducing PLATFORM_SendHciMessageAlt() alternative API.
- [wireless_mcu][wireless_nbu] Added NBU2Host event manager for status indications to host (Information, Warning, Error) sent over RPMSG.
- [wireless_mcu] Added a call to PLATFORM_IcsRxWorkHandler() within PLATFORM_NbuApiReq() for baremetal applications to prevent potential deadlocks.

Minor Changes

- [wireless_mcu] Fixed variable underflow issue in PLATFORM_RemoteActiveRel().
- [SecLib_RNG] Fixed escaping local HashKeyBuffer address issue and added missing cast in RNG_GetTrueRandomNumber() function.
- [Common] Fixed heap memory manager return values and added missing include to fwk_freertos_utils.h.
- [rw61x] Prevented array out of bounds in PLATFORM_RegisterRtcHandle().
- [FSCI] Fixed memory leak in FSCI module.
- [NVM] Enhanced debug facilitation by restricting variable scope, assigning return statuses to variables, and fixing display format in NV_ShowFlashTable().
- [wireless_mcu] Added new chip revision A2.1 support in PLATFORM_SendChipRevision() API.
- [kw47_mcxw72] Implemented BLE BD address retrieval from IFR memory with fallback to OUI + RNG.
- [DBG] Added ThreadX support to fault handlers and reworked fault handler structure with dedicated RTOS files.
- [DBG][Common] Added NBU debug support on host side to detect faults and system errors, with debug info extraction capability (limited to MCXW72/KW47).
- [Common] Platform CMake rework and Kconfig renaming, removing unneeded checks and renaming PRJSEG platform Kconfigs to COMPONENT.
- [mcxw23] Added experimental SecLib PSA support with additional configuration for MBEDTLS_ECP_C and MBEDTLS_BIGNUM_C.
- [wireless_mcu] Cleaned CMakeLists.txt to avoid wrong inclusions of files and folders from incorrect platforms.
- [wireless_mcu][wireless_nbu] Added NBU2Host warning when 32MHz crystal is unready on low power exit.

- [wireless_mcu][ot] Introduced gPlatformUseOuiFromIfr to use OUI from IFR for the extended address (disabled by default). When enabled and IFR is not blank, copies first three bytes to OUI field of extended address, otherwise uses static OUI as fallback.
- [General] Removed useless warning about TSTMR_CLOCK_FREQUENCY_MHZ definition.
- [General] Updated framework license and SBOM for 25.09 RFP release.
- [wireless_mcu] Fixed unused variable warning when gPlatformIcsUseWorkqueueRxProcessing_d and gPlatformHciUseWorkqueueRxProcessing_dare disable

7.0.3 revB mcux SDK 25.09.00

Major Changes

- [wireless_mcu] Adjusted default value of BOARD_RADIO_DOMAIN_WAKE_UP_DELAY from 0x16 to 0x10 to address stability issues observed with the previous setting. This change enhances system reliability but will reduce low-power performance.

Minor Changes (bug fixes)

- [Common] Added MDK compatibility for the errno framework header.
- [mcxw23] Implemented missing PLATFORM_OtaClearBootInterface() API.
- [mcxw23] Refactored fwk_platform.c to separate BLE-specific logic into fwk_platform_ble.c.
- [OTA] Corrected definition of gEepromParams_WriteAlignment_c flag for mcxw23
- [OTA] Enabled calling OTA_GetImgState() prior to OTA_Initialize().
- [wireless_mcu] Fixed PLATFORM_IsExternalFlashSectorBlank() to check the entire sector instead of just one page.
- [mcxw23] Added support for OTA using external flash.
- [mcxw23] Introduced PLATFORM_GetRadioIdleDuration32K() to estimate time until next radio event.
- [OTA] Removed gUseInternalStorageLink_d linker flag definition when external OTA storage is used.
- [mcxw23] Extended CopyAndReboot() to support external flash OTA.
- [wireless_mcu] Resolved counter wrap issue in PLATFORM_GetDeltaTimeStamp().
- [kw43_mcxw70] Defined LPTMR frequency constants in fwk_platform_definitions.h.
- [kw47_mcxw72] Updated shared memory allocation for RPMsg adapter.
- [mcxw23] Implement PLATFORM_IsExternalFlashBusy() API.
- [kw45_mcxw71][kw47_mcxw72] Moved RAM bank definitions from the connectivity framework to device-specific definitions.

7.0.3 revA mcux SDK 25.09.00

Major Changes

- [wireless_nbu] Enhanced XTAL32M trimming handling: updates are applied when requested by the application core and the NBU enters low-power mode, ensuring no interference from ongoing radio activity. Introduced new APIs to lock (PLATFORM_LockXtal32MTrim()) and unlock XTAL32M

(PLATFORM_UnlockXtal32MTrim()) trimming updates using a counter-based mechanism. Also added a reset API (PLATFORM_ResetContext()) for platform-specific variables (currently limited to the trimming lock).

- [wireless_mcu] Introduced a new API, PLATFORM_SetLdoCoreNormalDriveVoltage(), to enable support for NBU clock frequency at 64 MHz, as required by BLE channel sounding applications.
- [wireless_mcu][wireless_nbu] Increased delayLpoCycle default from 2 to 3 to address link layer instabilities in low-power NBU use cases. Adjusted BOARD_RADIO_DOMAIN_WAKE_UP_DELAY from 0x10 to 0x16 to balance power consumption and stability. NBU may malfunction if delayLpoCycle (or BOARD_LL_32MHz_WAKEUP_ADVANCE_HSLOT) is set to 2 while BOARD_RADIO_DOMAIN_WAKE_UP_DELAY is 0x16.

Minor Changes (bug fixes)

- [WorkQ] Increased stack size when RNG use mbedtls port and coverage is enabled.
- [FSCI] Resolved an issue where messages remained unprocessed in the queue by ensuring OSA_EventSet() is triggered when pending messages are detected.
- [OTA] Fixed a bug in in OTA_PullImageChunk() that prevented retrieval of data previously received via OTA_PushImageChunk() when still buffered in RAM during posted operations.
- [OTA] Various MISRA and coverity fixes.
- [mcxw23] Fixed an unused variable warning in PLATFORM_RegisterNbuTemperatureRequestEventCb() API.
- [SFC] Remove obsolete flag gNbuJtagCapability.
- [wireless_mcu] Introduced new API PLATFORM_GetRadioIdleDuration32K(). Deprecated PLATFORM_CheckNextBleConnectivityActivity() API.
- [mcxw23] Aligned platform-specific implementations with the corresponding prototypes defined in wireless_mcu.
- [DBG] Cleaned up fwk_fault_handler.c.

7.0.2 RFP mcux SDK 25.06.00

Major Changes

- [wireless_mcu][wireless_nbu] Introduced PLATFORM_Get32KTimeStamp() API, available on platforms that support it.
- [RNG] Switched to using a workqueue for scheduling seed generation tasks.
- [Sensors] Integrated workqueue to trigger temperature readings on periodic timer expirations.
- [wireless_nbu] Removed outdated configuration files from wireless_nbu/configs.
- [SecLib_RNG][PSA] Added a PSA-compliant implementation for SecLib_RNG. This is an experimental feature and should be used with caution.
- [wireless_mcu][wireless_nbu] Implemented PLATFORM_SendNBUXtal32MTrim() API to transmit XTAL32M trimming values to the NBU.

Minor Changes (bug fixes)

- [MWS] Migrated the Mobile Wireless Standard (MWS) service to the public repository. This service manages coexistence between connectivity protocols such as BLE, 802.15.4, and GenFSK.
- [HWParameter][NVM][SecLib_RNG][Sensors] Addressed various MISRA compliance issues across multiple modules.
- [Sensors] Applied a filtering mechanism to temperature data measured by the application core before forwarding it to the NBU, improving data reliability.
- [Common] Relocated the GetPowerOfTwoShift() function to a shared module for broader accessibility across components.
- [RNG] Resolved inconsistencies in RNG behavior when using the fsl_adapter_rng HAL by aligning it with other API implementations.
- [SecLib] Updated the AES CMAC block counter in AES_128_CMAC() and AES_128_CMAC_LsbFirstInput() to support data segments larger than 4KB.
- [SecLib] Utilized sss_sscp_key_object_free() with kSSS_keyObjFree_KeysStoreDefragment to avoid key allocation failures.
- [MCXW23] Removed redundant NVIC_SetPriority() call for the ctimer IRQ in the platform file, as it's already handled by the driver.
- [WorkQ] Increased workqueue stack size to accommodate RNG usage with mbedtls.
- [wireless_mcu][ot] Suppressed chip revision transmission when operating with nbu_15_4.
- [platform][mflash] Ensured proper address alignment for external flash reads in PLATFORM_ReadExternalFlash() when required by platform constraints.
- [RNG] Corrected reseed flag behavior in RNG_GetPseudoRandomData() after reaching gRngMaxRequests_d threshold.
- [platform][mflash] Fixed uninitialized variable issue in PLATFORM_ReadExternalFlash().
- [platform][wireless_nbu] Fixed an issue on KW47 where PLATFORM_InitFro192M incorrectly reads IFR1 from a hardcoded flash address (0x48000), leading to unstable FRO192M trimming. The function is now conditionally compiled for KW45 only.

7.0.2 revB mcux SDK 25.06.00

Major Changes

- [RNG][wireless_mcu][wireless_nbu] Rework RNG seeding on NBU request
- [wireless_mcu] [LowPower] Add gPlatformEnableFro6MCalLowpower_d macro to enable FRO6M frequency verification on exit of Low Power
  - add PLATFORM_StartFro6MCalibration() and PLATFORM_EndFro6MCalibration() new function for FRO6M calibration (6MHz or 2Mhz) on wake-up from low power mode.
  - Enabled by default in fwk_config.h
- [wireless_nbu][LowPower] Clear pending interrupt status of the systick before going in low-power - Reduce NBU active time
- [wireless_nbu] Fix impossibility to go to WFI in combo mode (15.4/BLE)
- [wireless_mcu] Implement XTAL32M temperature compensation mechanism. 2 new APIs:
  - PLATFORM_RegisterXtal32MTempCompLut(): register the temperature compensation table for XTAL32M.

  - PLATFORM_CalibrateXtal32M(): apply XTAL32M temperature compensation depending on current temperature.
- [Sensors][wireless_mcu] Add support for periodic temperature measurement. new API:
  - SENSORS_TriggerTemperatureMeasurementUnsafe(): to be called from Interrupt masked critical section, from ISR or when scheduler is stopped
- [SFC] Change default maximal ppm target of the SFC algorithm from 200 to 360ppm. Impact the SFC algorith of kw45 and mcxw71 platforms, 360ppm was already the default setting for kw47 and mcxw72 platforms

Minor Changes (bug fixes)

- [DBG] Fix FWK_DBG_PERF_DWT_CYCLE_CNT_STOP macro
- [wireless_nbu] Add gPlatformIsNbu_d compile Macro set to 1
- [wireless_nbu][ics] gFwkSrvHostChipRevision_c can be processed in the system workqueue
- [kw45_mcxw71][kw47_mcxw72]
  - Remove LTC dependency from platform in kconfig
  - gPlatformShutdownEccRamInLowPower moved from fwk_platform_definition.h to fwk_confg.h as this is a configuration flag.
- [wireless_mcu][sensors] Rework and remove unnecessary ADC APIs
- [wireless_nbu] Add PLATFORM_GetMCUUid() function from Chip UID
- [SecLib] Change AES_MMO_BlockUpdate() function from private to public for zigbee.

7.0.2 revA mcux SDK 25.06.00 Supported platforms:

- Same as 25.03.00 release

Major Changes

- [KW45/MCXW71] HW parameters placement now located in IFR section. Flash storage is not longer used:
  - Compilation: Macro gHwParamsProdDataPlacement_c changed from gHwParamsProdDataMainFlash2IfrMode_c to gHwParamsProdDataIfrMode_c
- [KW47] NBU: Add new fwk_platform_dcdc.[ch] files to allow DCDC stepping by using SPC high power mode. This requires new API in board_dcdc.c files. Please refer to new compilation MACROs gBoardDcdcRampTrim_c and gBoardDcdcEnableHighPowerModeOnNbu_d in board_platform.h files located in kw47evk, kw47loc, frdmmcxw72 board folders.
- [KW45/MCXW71/KW47/MCXW72] Trigger an interrupt each time App core calls PLATFORM_RemoteActiveReq() to access NBU power domain in order to restart NBU core for domain low power process

Minor Changes (bug fixes)

Services

- [SecLib_RNG]
  - Rename mSecLibMutexId mutex to mSecLibSssMutexId in SecLib_sss.c
  - Remove MEM_TRACKING flag from RNG.c

  - Implement port to fsl_adapter_rng.h API using gRngUseRngAdapter_c compil Macro from RNG.c
  - Add support for BLE debug Keys in SecLi and SecLin_sss.c with gSecLibUseBleDebugKeys_d - for Debug only
- [FSCI] Add queue mechanism to prevent corruption of FSCI global variableAllow the application to override the trig sample number parameter when gFsciOverRpmsg_c is set to 1
- [DBG][btsnoop] Add a mechanism to dump raw HCI data via UART using SBTSNOOP_MODE_RAW
- [OTA]
  - OtaInternalFlash.c: Take into account chunks smaller than a flash phrase worth
  - fwk_platform_ot.c: dependencies and include files to gpio, port, pin_mux removed

Platform specific

- [kw45_mcxw71][kw47_mcxw72]
  - fwk_platform_reset.h : add compil Macro gUseResetByLvdForce_c and gUseResetByDeepPowerDown_c to avoid compile the code if not supported on some platforms
  - New compile Flag gPlatformHasNbu_d
  - Rework FRO32K notification service for MISRA fix

7.0.1 RFP mcux SDK 25.03.00 Supported platforms:

- KW45x, KW47x, MCXW71, MCXW72, K32W1x
- RW61x
- RT595, RT1060, RT1170
- MCXW23

Minor Changes (bug fixes)

- [General] Various MISRA/Coverity fixes in framework: NVM, RNG, LowPower, SecLib and platform files

Services

- [SecLib_RNG] fix return status from RNG_GetTrueRandomNumber() function: return correctly gRngSuccess_d when RNG_entropy_func() function is successful
- [SFC] Allow the application to override the trig sample number parameter
- [Settings] Re-define the framework settings API name to avoid double definition when gSettingsRedefineApiName_c flag is defined

Platform specific

- [wireless_mcu] fwk_platform_sensors update :
  - Enable temperature measurement over ADC ISR
  - Enable temperature handling requested by NBU
- [wireless_mcu] fwk_platform_lcl coex config update for KW45

- [kw47_mcxw72] Change the default ppm_target of SFC algorithm from 200 to 360ppm

7.0.1 revB mcux SDK 25.03.00 Supported platforms:

- KW45x, KW47x, MCXW71, MCXW72, K32W1x
- RW61x
- RT595, RT1060, RT1170
- MCXW23

Minor Changes (bug fixes)

General

- [General] Various MISRA/Coverity fixes in framework: NVM, RNG, LowPower, FunctionLib and platform files

Services

- [SecLib_RNG] AES-CBC evolution:
  - added AES_CBC_Decrypt() API for sw, SSS and mbedtls variants.
  - Made AES-CBC SW implementation reentrant avoiding use of static storage of AES block.
  - fixed SSS version to update Initialization Vector within SecLib, simplifying caller's implementation.
  - modified AES_128_CBC_Encrypt_And_Pad() so as to avoid the constraint mandating that 16 byte headroom be available at end of input buffer.
- [SecLib_RNG] RNG modifications:
  - RNG_GetPseudoRandomData() could return 0 in some error cases where caller expected a negative status.
    - Explicited RNG error codes
    - Added argument checks for all APIs and return gRngBadArguments_d (-2) when wrong
    - added checks of RNG initalization and return gRngNotInitialized_d (-3) when not done
    - fixed correcteness of RNG_GetPrngFunc() and RNG_GetPrngContext() relative to API description.
    - Added RNG_DeInit() function mostly for test and coverage purposes.
    - Improved RNG description in README.md
    - Unified the APIs behaviour between mbedtls and non mbedtls variants.
  - RNG/mbedtls : Prevent RNG_Init() from corrupting RNG entropy context if called more than once.
  - RNG/mbedtls: fixed RNG_GetTrueRandomNumber() to return a proper mbedtls_entropy_func() result.
  - Use defragmetation option when freeing key object in SecLib_sss to avoid leak in S200 memory
  - Add new API ECP256_IsKeyValid() to check whether a public key is valid

- [OtaSupport] Update return status to OTA_Flash_Success when success at the end of InternalFlash_WriteData() and InternalFlash_FlushWriteBuffer() APIs
- [WorQ] Implementing a simple workqueue service to the framework
- [SFC] Keep using immediate measurement for some measurement before switching to configuration trig to confirm the calibration made
- [DBG] Adding modules to framework DBG :
  - sbtsnoop
  - SWO
- [Common] Fix HAL_CTZ and HAL_RBIT IAR versions
- [LowPower] Fix wrong tick error calculation in case of infinite timeout
- [Settings] Add new macro gSettingsRedefineApiName_c to avoid multiple definition of settings API when using connectivity framework repo

Platform specific

- [KW47/MCXW72] Change xtal cload default value from 4 to 8 in order to increase the precision of the link layer timebase in NBU
- [wireless_mcu] [wireless_nbu] Use new WorkQ service to process framework intercore messages
- [rw61x] Fix HCI message sending failure in some corner case by releasing controller wakes up after that the host has send its HCI message
- [MCXW23] Adding the initial support of MCXW23 into the framework

7.0.0 mcux SDK 24.12.00 **Supported platforms:**

- KW45x, KW47x, MCXW71, MCXW72, K32W1x
- RW61x
- RT595, RT1060, RT1170

Minor Changes (bug fixes)

Platform specific

- [RW61X]
  - Add MCUX_COMPONENT_middleware.wireless.framework.platform.rng to the platform to fix a warning at generation
  - Retrieve IEEE 64 bits address from OTP memory
- [KW45x, MCXW71x, KW47x, MCXW72x]
  - Ignore the secure bit from RAM addresses when comparing used ram bank in bank retention mechanism
  - Add gPlatformNbuDebugGpioDAccessEnabled_d Compile Macro (enabled by default). Can be used to disable the NBU debug capability using IOs in case Trustzone is enabled (“PLATFORM_InitNbu()‘ code executed from unsecure world).
  - Fix in NBU firmware when sending ICS messages gFwkSrvNbuApiRequest_c (from controller_api.h API functions)

Services

- [OTA]
  - Add choice name to OtaSupport flash selection in Kconfig
- [NVM]
  - Add gNvmErasePartitionWhenFlashing_c feature support to gcc toolchain
- [SecLib_RNG]
  - Misra fixes

7.0.0 revB mcux SDK 24.12.00 Supported platforms: KW45x, KW47x, MCXW71, MCXW72, K32W1x, RW61x, RT595, RT1060, RT1170

Major Changes (User Applications may be impacted)

- mcux github support with cmake/Kconfig from sdk3 user shall now use CmakeLists.txt and Kconfig files from root folder. Compilation should be done using west build command. In order to see the Framework Kconfig, use command >west build -t guiconfig
- Board files and linker scripts moved to examples repository

Bugfixes

- [platform lowpower]
  - Entering Deep down power mode will no longer call PLATFORM_EnterPowerDown(). This API is now called only when going to Power down mode

Platform specific

- [KW47/MCXW72]: Early access release only
  - Deep sleep power mode not fully tested. User can experiment deep sleep and deep down modes using low power reference design applications
  - XTAL32K-less support using FRO32K not tested
- [KW45/MCXW71/K32W148]
  - Deep sleep mode is supported. Power down mode is supported in low power reference design applications as experimental only
  - XTAL32K-less support using FRO32K is experimental - FRO32K notifications callback is debug only and should not be used for mass production firmware builds

Minor Changes (no impact on application)

- Overall folder restructuring for SDK3
  - [Platform]:
    - Rename platform_family from connected_mcu/nbu to wireless_mcu/nbu
    - platform family have now a dedicated fwk_config.h, rpmsg_config.h and SecLib_mbedtls_config.h
  - [Services]
    - Move all framework services in a common directory “services/”

7.0.0 revA: KW45/KW47/MCX W71/MCX W72/K32W148

Experimental Features only

- Power down on application power domain: Some tests have shown some failure. Power consumption higher than Deep Sleep. => This feature is not fully supported in this release
- XTAL32K less board with FRO32K support: Some additional stress tests are under progress.
- FRO32K notifications callback is for debug only and shall not be used for production. User shall not execute long processing (such as PRINTF) as it is executed in ISR context.

Main Changes

- Cmake/Kconfig support for SDK3.0
- [Sensors] API renaming:
  - SENSORS_InitAdc() renamed to SENSORS_Init()
  - SENSORS_DeinitAdc() remamed to SENSORS_Deinit()
- [HWparams]
  - Repair PROD_DATA sector in case of ECC error (implies loss of previous contents of sector)
- [NVM] Linker script modification for armgcc whenever gNvTableKeptInRam_d option is used:
  - placement of NVM_TABLE_RW in data initialized section, providing start and end address symbols. For details see NVM_Interface.h comments.
- [OtaSupport]
  - OTA_Initialize(): now transitions the image state from RunCandidate to Permanent if not done by the application. OTA module shall always be initialized on a Permanent image, this change ensures it is the case.
  - OTA_MakeHeadRoomForNextBlock(): now erases the OTA partition up to the image total size (rounded to the sector) if known.

Minor changes

- [Platform]
  - Updated macro values: -kw47: BOARD_32MHZ_XTAL_CDAC_VALUE from 12U to 16U, BOARD_32MHZ_XTAL_ISEL_VALUE from 7U to 11U, BOARD_32KHZ_XTAL_CLOAD_DEFAULT from 8U to 4U, BOARD_32KHZ_XTAL_COARSE_ADJ_DEFAULT from 1U to 3U
    - MCX W72 (low-power reference design applications only): BOARD_32MHZ_XTAL_CDAC_VALUE from 12U to 10U, BOARD_32MHZ_XTAL_ISEL_VALUE from 7U to 11U, BOARD_32KHZ_XTAL_CLOAD_DEFAULT from 8U to 4U, BOARD_32KHZ_XTAL_COARSE_ADJ_DEFAULT from 1U to 3U
  - New PLATFORM_RegisterNbuTemperatureRequestEventCb() API: register a function callback when NBU request new temperature measurement. API provides the interval request for the temperature measurement
  - Update PLATFORM_IsNbuStarted() API to return true only if the NBU firmware has been started.
- [platform lowpower]

  - Move RAM layout values in fwk_platform_definition.h and update RAM retention API for KW47/MCXW72

Bugfixes

- [OtaSupport]
  - OTA_MakeHeadRoomForNextBlock(): fixed a case where the function could try to erase outside the OTA partition range.

6.2.4: KW45/K32W1x/MCXW71/RX61x SDK 2.16.100 This release does not contain the changes from 6.2.3 release.

This release contains changes from 6.2.2 release.

Main Change

- armgcc support for Cmake sdk2 support and VS code integration

Minor changes

- [NBU]
  - Optimize some critical sections on nbu firmware
- [Platform]
  - Optimize PLATFORM_RemoteActiveReq() execution time.

6.2.3: KW47 EAR1.0 Initial Connectivity Framework enablement for KW47 EAR1.0 support.

New features

- OpenNBU feature : nbu_ble project is available for modification and building

Supported features

- Deep sleep mode

Unsuported features

- Power down mode
- FRO32K support (XTAL32K less boards)

Main changes

- [NBU]
  - LPTMR2 available and TimerManager initialization with Compile Macro: gPlatformUseLptmr_d
  - NBU can now have access to GPIOD
  - SW RNG and SW SecLib ported to NBU (Software implementation only)
- [RNG]
  - Obsoleted API removed : FWK_RNG_DEPRECATED_API

  - RNG can be built without SecLib for NBU, using gRngUseSecLib_d in fwk_config.h
  - Some API updates:
    * RNG_IsReseedneeded() renamed to RNG_IsReseedNeeded,
    * RNG_TriggerReseed() renamed to RNG_NotifyReseedNeeded(),
    * RNG_SetSeed() and RNG_SetExternalSeed() return status code.
  - Optimized Linear Congruential modulus computation to reduce cycle count.

Minor changes

- [NVM]
  - Optimize NvIsRecordErased() procedure for faster garbage collection
  - MISRA fix : Remove externs and weaks from NVM module - Make RNG and timer manager dependencies conditional
- [Platform]
  - Allow the debugger to wakeup the KW47/MCXW72 target

6.2.2: KW45/K32W1 MR6 SDK 2.16.000

Experimental Features only:

- Power down on application power domain : Some tests have shown some failure. Power consumption higher than Deep Sleep. => This feature is not fully supported in this release
- XTAL32K less board with FRO32K support : Some additional stress tests are under progress.
- FRO32K notifications callback is for debug only and shall not be used for production. User shall not execute long processing (such as PRINTF) as it is executed in ISR context.

Changes

- [Board] Support for freedom board FRDM-MCX W7X
- [HWparams]
  - Support for location of HWParameters and Application Factory Data IFR in IFR1
  - Default is still to use HWparams in Flash to keep backward compatibility
- [RNG]: API updates:
  - New APIS RNG_IsReseedneeded(), RNG_SetSeed() to provide See to PRNG on NBU/App core - See BluetoothLEHost_ProcessIdleTask() in app_conn.c
  - New APIs RNG_SetExternalSeed() : User can provide external seed. Typically used on NBU firmrware for App core to set a seed to RNG. RNG_TriggerReseed() : Not required on App core. Used on NBU only.
- [NVS] Wear statistics counters added - Fix nvs_file_stat() function
- [NVM] fix Nv_Shutdown() API
- [SecLib] New feature AES MMO supported for Zigbee

6.2.2: RW61x RFP4 SDK 2.16.000

- [Platform] Support Zigbee stack
- [OTA] Add support for RW61x OTA with remap feature.

  - Required modifications to prevent direct access to flash logical addresses when remap is active.
  - Image trailers expected at different offset with remap enabled (see gPlatformMcuBootUseRemap_d in fwk_config.h)
  - fixed image state assessment procedure when in RunCandidate.
- [NVS] Wear statistics counters added
- [SecLib] New feature AES MMO supported for Zigbee
- [Misra] various fixes

6.2.1: KW45/K32W1 MR5 SDK 2.15.000 Experimental Features only:

- Power down on application power domain : Some tests have shown some failure. This feature is not fully supported in this release
- XTAL32K less board with FRO32K support : Some additional stress tests are under progress. Timing variation of the timebase are being analyzed

Major changes

- [RNG]: API updates
  - New compile flag to keep deprecated API: FWK_RNG_DEPRECATED_API
  - change return error code to int type for RNG_Init(), RNG_ReInit()
  - New APIs RNG_GetTrueRandomNumber(), RNG_GetPseudoRandomData()
- [Platform]
  - fwk_platform_sensors
    * Change default temperature value from -1 to 999999 when unknown
  - fwk_platform_genfsk
    * rename from platform_genfsk.c/h to fwk_platform_genfsk.c/h
  - platform family
    * Rename the framework platform folder from kw45_k32w1 to connected_mcu to support other platform from the same family
  - fwk_platform_intflash
    * Moved from fwk_platform files to the new fwk_platform_intflash files the internal flash dependant API
- [NBU]
  - BOARD_LL_32MHz_WAKEUP_ADVANCE_HSLOT changed from 2 to 3 by default
  - BOARD_RADIO_DOMAIN_WAKE_UP_DELAY changed from 0x10 to 0x0F
- [gcc linker]
  - Exclude k32w1_nbu_ble_15_4_dyn.bin from .data section

Minor Changes

- [Platform]
  - PLATFORM_GetTimeStamp(0 has an important fix for reading the Timestamp in TSTMR0

  - New API PLATFORM_TerminateCrypto(), PLATFORM_ResetCrypto() called from SecLib for lowpower exit
  - Fix when enable fro debug callback on nbu
- [DBG]
  - SWO
    * Add new files fwk_debug_swo.c/h to use SWO for debug purpose
    * Two new flags has been added:
      · BOARD_DBG_SWO_CORE_FUNNEL to chose on which core you want to use SWO
      · BOARD_DBG_SWO_PIN_ENABLE to enable SWO on a pin
- [NVS]
  - Add support of NVS and Settings in framework
- [NBU]
  - Fix power down issues and reduce critical section on NBU side:
    * new API PLATFORM_RemoteActiveReqWithoutDelay() called from NBU functions where waiting delay is not required
    * Increase delay needed in power down for OEM part to request the SOC to be active
    * Remove unnecessary code to PLATFORM_RemoteActiveReqWithoutDelay() from PLATFORM_HciRpmsgRxCallback()
    * Improve nbu memory allocation failure debug messages
- [SDK]
  - Multicore: remove critical section in HAL_RpmsgSendTimeout() (only required in FPGA HDI mode)
  - Flash drivers: update for ECC detection
- [Platform]
  - fwk_platform_sensors
    * Fix temperature reporting to NBU
  - fwk_platform_extflash
    * Align .c and .h prototype of PLATFORM_ExternalFlashAreaIsBlank() function
- [NVM]
  - Keep Mutex in NvModuleDeInit(). In Bare metal OS, Mutex can not be destroyed
  - New API NvRegisterEccFaultNotificationCb() to register Notification callback when Ecc error happens in FileSystem
- [MISRA] fixes
  - SecLib_sss.c: ECDH_P256_ComputeDhKey()
  - fwk_platform_extflash.c: PLATFORM_IsExternalFlashPageBlank()
  - fwk_fs_abstraction.c: Various fixes
- [HWparams]
  - Fix on if condition when gHwParamsProdDataPlacementLegacy2IfrMode_c mode is selected
- [OTA]

  - Enable gOtaCheckEccFaults_d by default to avoid bus in case of ECC error during OTA
  - Fix OTA partition overflow during OTA stop and resume transfer
- [BOARD]
  - Place code button or led specific under correct defines in board_comp.c/h
  - Bring back MACROs BOARD_INITRFSWITCHCONTROLPINS in pin_mux header file of the loc board
- [SecLib]
  - Add some undefinition in SecLib_mbedtls_config as new dependency has been added in mbedtls repo:
    * MBEDTLS_SSL_CBC_RECORD_SPLITTING, MBEDTLS_SSL_PROTO_TLS1, MBEDTLS_SSL_PROTO_TLS1_1
- [FRO32K]
  - FRO32K notification callback PLATFORM_FroDebugCallback_t() has new parameter to report he fro_trim value
  - maxCalibrationIntervalMs value can be provided to NBU using PLATFORM_FwkSrvSetRfSfcConfig()
- [Sensors]
  - fix: PLATFORM_GetTemperatureValue() shall have NBU started to send temperature to NBU

6.2.1: RW61x RFP3

- [NVS]
  - Add support of NVS and Settings in framework
- [MISRA] fixes
  - board_lp.c BOARD_UninitDebugConsole() and BOARD_ReinitDebugConsole()
  - fwk_platform_ble.c: Various fixes
- [OTA]
  - Fix OTA partition overflow during OTA stop and resume transfer

6.2.0: RT1060/RT1170 SDK2.15 Major

6.1.8: KW45/K32W1 MR4

- [BOARD PLATFORM]
  - Move gBoardUseFro32k_d to board_platform.h file
  - Offer the possibility to change the source clock accuracy to gain in power consumption
- [BOARD LP]
  - Move PLATFORM_SetRamBanksRetained() at end of BOARD_EnterLowPowerCb() in case a memory allocation is done previously in this function
  - fix low power, increase BOARD_RADIO_DOMAIN_WAKE_UP_DELAY from 0 to 0x10 - Skip this delay when App requesting NBU wakeup
- [PLATFORM]

  - fwk_platform_ble.c/h: New timestamp API that returns the difference between the current value of the LL clock and the argument of the function
  - fwk_platform.c/h:
    * New PLATFORM_EnableEccFaultsAPI_d compile flag: Enable APIs for interception of ECC Fault in bus fault handler
    * New gInterceptEccBusFaults_d compile flag: Provide FaultRecovery() demo code for bus fault handler to Intercept bus fault from Flash Ecc error
- [LOC]
  - Incorrect behavior for set_dtest_page (DqTEST11 overridden)
  - Fix SW1 button wake able on Localization board
  - Fix yellow led not properly initialized
  - Format localization pin_mux.c/h files
- [Inter Core]
  - Affect values to enumeration giving the inter core service message ids
  - Shared memory settings shared between both cores
  - Add callback to register when NBU has unrecoverable Radio issue
- [NVM]
  - Add NV_STORAGE_MAX_SECTORS, NV_STORAGE_SIZE as linker symbol for alignment with other toolchain
  - ECC detection and recovery. New gNvSalvageFromEccFault_d and gNvVerifyReadBackAfterProgram_d compile flags. Please refer to ECC Fault detection section in README.md file located in NVM folder
- [OTA]
  - Prevent bus fault in case of ECC error when reading back OTA_CFR update status (disable by default)
- [SecLib]
  - Shared mutex for RNG and SecLib as they share same hardware resource
- [Key storage]
  - Fix to ignore the garbage at the end of buffers
  - Detect when buffers are too small in KS_AddKey() functions
- [FileCache]
  - Fix deadlock in Filecache FC_Process()
- [SDK]
  - Applications: remove definition of stack location and use default from linker script, fix warmboot stack in freertos at 0x20004000
  - Memory Manager Light:
    * fix Null pointer harfault when MEM_STATISTICS_INTERNAL enable
    * Fix MemReinitBank() on wakeup from lowpower when Ecc banks are turned off

6.1.7: KW45/K32W1 MR3

- [OTA]
  - New API OTA_SetNewImageFlagWithOffset()
  - Fix StorageBitmapSize calculation
  - OTA clean up: Removed OTA_ValidateImage()
- [Low Power]
  - New linker Symbol m_lowpower_flag_start in linker file.
    - Flag is used to indicate NBU that Application domain goes to power down mode. Keep this flag to 0 if only Deep sleep is supported
    - This flag will be set to 1 if Application domain goes to power down mode
  - Re-introduce PWR_AllowDeviceToSleep()/PWR_DisallowDeviceToSleep(), PWR_IsDeviceAllowedToSleep() API
  - Implement tick compensation mechanism for idle hook in a dedicated freertos utils file fwk_freertos_utils.[ch], new functions: FWK_PreIdleHookTickCompensation() and FWK_PostIdleHookTickCompensation
  - Rework timestamping on K4W1
    - PLATFORM_GetMaxTimeStamp() based on TSTMR
    - Rename PLATFORM_GetTimestamp() to PLATFORM_GetTimeStamp()
    - Update PLATFORM_Delay(): Rework to use TSTMR instead of LPTMR for platform_delay
    - Update PLATFORM_WaitTimeout(): Fixed a bug in PLATFORM_WaitTimeout() related to timer wrap
    - Add PLATFORM_IsTimeoutExpired() API
  - Fix race condition in PWR_EnterLowPower(), masking interrupts in case not done at upper layer
  - Low power timer split in new files fwk_platform_lowpower_timer.[ch]
  - New PWR_systicks_bm.c file for bare metal usage: implement SysTick suspend/resume functionality, New weak PWR_SysTicksLowPowerInit()
- [FRO32K]
  - Improve FRO32K calibration in NBU
  - create PLATFORM_InitFro32K() to initialize FRO32K instead of XTAL32K (to be called from hardware_init())
  - update FRO32K README.md file in SFC module
  - Debug:
  - Add Notification callback feature for SFC module FRO32K
  - Linker script update to support m_sfc_log_start in SMU2
- [SecLib]
  - Remove gSecLibSssUseEncryptedKeys_d compile option, split Secure/Unsecure APIs
  - RNG update to use same mutex than SecLib
  - Fix AES_128_CBC_Encrypt_And_Pad length
  - Implement RNG_ReInit() for lowpower
  - Fix issue in ECDH_P256_GenerateKeys() when waking up from power down

  - Call CRYPTO_ELEMU_reset() from SecLib_reInit() for power down support
- [BOARD]
  - Create new board_platform.h file for all Board characteristics settings (32Mhz XTAL, 32KHZ XTAL, etc..)
  - TM_EnterLowpower() TM_EnterLowpower() to be called from LP callbacks
  - Support Localization boards, Only BUTTON0 supported
    - New compile flag BOARD_LOCALIZATION_REVISION_SUPPORT
    - New pin_mux.[ch] files
  - Offer the possibility to override CDAC and ISEL 32MHz settings before the initialization of the crystal in board_platform.h
    - new BOARD_32MHZ_XTAL_CDAC_VALUE, BOARD_32MHZ_XTAL_ISEL_VALUE
    - BOARD_32MHZ_XTAL_TRIM_DEFAULT obsoleted
- [NVM file system]
  - Look ahead in pending save queue - Avoid consuming space to save outdated record
  - Fix NVM gNvDualImageSupport feature in NvIsRecordCopied
- [Inter Core]
  - Change PLATFORM_NbuApiReq() API return parameters granularity from uint32 to uint8
  - MAX_VARIANT_SZ change from 20 to 25
  - Set lp wakeup delay to 0 to reduce time of execution on host side, NBU waits XTAL to be ready before starting execution
  - Update inter core config rpmsg_config.h
  - Add timeout to while loops that relies on hardware in RemoteActiveReq(), Application can register Callbacks when timeout
  - Return non-0 status when calling PLATFORM_FwkSrvSendPacket when NBU non started
  - Let PLATFORM_GetNbuInfo return -10 if response not received on timeout - Doxygen platform_ics APIs
- [HW params]
  - New compile Macro for HW params placement in IFR - Save 8K in FLash: gHwParamsProdDataPlacement_c . 3 modes:
  - Legacy placement, move from legacy to IFR, IFR only placement
  - New compile Macro for Application data to be stored with HW params (in shared flash sector): gHwParamsAppFactoryDataExtension_d, New APIs:
    - Nv_WriteAppFactoryData(), Nv_GetAppFactoryData()
  - See HWParameter.h
- [Platform]
  - Implement PLATFORM_GetIeee802_15_4Addr() API in fwk_platform_ot.c - New gPlatformUseUniqueDeviceIdFor15_4Addr_d compile Macro
  - Wakeup NBU domain when reading RADIO_CTRL UID_LSB register in PLATFORM_GenerateNewBDAddr()
- [Reset]
  - New reset Implementations using Deep power down mode or LVD:

    * new files fwk_platform_reset.[ch]
    * new APIs: PLATFORM_ForceDeepPowerDownReset(), PLATFORM_ForceLvdReset() + reset on ext pins
    * new compile flags: gAppForceDeepPowerDownResetOnResetPinDet_d and gAppForceLvdResetOnResetPinDet_d to reset on external pins
- [FSCI]
  - fix when gFsciRxAck_c enabled
  - integrate new reset APIs

6.1.4: RW610/RW612 RFP1

- [Low Power]
  - Added support of low power for OpenThread stack.
  - Added PWR_AllowDeviceToSleep/PWR_DisallowDeviceToSleep/PWR_IsDeviceAllowedToSleep APIs.
- [platform]
  - Added PLATFORM_GetMaxTimeStamp API.
  - Fixed high impact Coverity.
- [FreeRTOS]
  - Created a new utilities module for FreeRTOS: fwk_freertos_utils.c/h.
  - Implemented a tick compensation mechanism to be used in FreeRTOS idle hook, likely around flash operations. This mechanism aims to estimate the number of ticks missed by FreeRTOS in case the interrupts are masked for a long time.

6.1.4: KW45/K32W1 MR2

- [Low power]
  - Powerdown mode tested and enabled on Low Power Reference Design applications
  - XTAL32K removal functionality using FRO32K, supported from NBU firmwares - limitation: Application domain supports Deep Sleep only (not power down)
  - NBU low power improvement: low power entry sequence improvement and system clock reduction to 16Mhz during WFI
  - Wake up time from cold boot, reset, power switch greatly improved. Device starts on FRO32K, switch to XTAL32K when ready if gBoardUseFro32k_d not set
  - Bug fixes:
    * Move PWR LowPower callback to PLATFORM layers
    * Fix wrong compensation of SysTicks
    * Reinit system clocks when exiting power down mode: BOARD_ExitPowerDownCb(), restore 96MHz clock is set before going to low power
    * Call Timermanager lowpower entry exit callbacks from PLATFORM_EnterLowPower()
    * Update PLATFORM_ShutdownRadio() function to force NBU for Deep power down mode
  - K32W1:

    * Support lowpower mode for 15.4 stacks

- [NVM]
  - New Compilation MACRO gNvDualImageSupport to support multiple firmware image with different register dataset
  - Change default configuration gNvStorageIncluded_d to 1, gNvFragmentation_Enabled_d to 1, gUnmirroredFeatureSet_d to TRUE
  - Some MISRA issues for this new configuration.
  - Remove deprecated functionality gNvUseFlexNVM_d
- [SecLib]
  - New NXP Ultrafast ecp256 security library:
    * New optimized API for ecdh DhKey/ecp256 key pair computation: Ecdh_ComputeDhKeyUltraFast(), ECP256_GenerateKeyPairUltraFast().
    * New macro gSecLibUseDspExtension_d.
    * Improved software version of Seclib with Ultrafast library for ECP256_LePointValid()
  - Bug fixes:
    * Share same mutex between Seclib and RNG to prevent concurrent access to S200
    * Optimized S200 re-initialization, restore ecdh key pair after power down
    * Fixed race condition when power down low power entry is aborted
    * Endianness function updates and clean up
- [OTA]
  - OTASupport improvements:
    * New API OTA_GetImgState(), OTA_UpdateImgState()
    * OTASupport and fwk_platform_extflash API updates for external flash: OTA_SelectExternalStoragePartition(), PLATFORM_IsExternalFlashSectorBlank(), PLATFORM_IsExternalFlashPageBlank(), PLATFORM_OtaGetOtaPartitionConfig()
    * Updated OtaExternalFlash.c, 2 new APIs in fwk_platform_extflash.c
    * Removed unused FLASH_op_type and FLASH_TransactionOpNode_t definitions from public API
    * Removed unused InternalFlash_EraseBlock() from OtaInternalFlash.c
- [NBU firmware]
  - Mechanism to set frequency constraint to controller from the host PLATFORM_SetNbuConstraintFrequency()
  - NbuInfo has one more digit in versionBuildNo field
- [Board]
  - Support Extflash low power mode, add BOARD_UninitExternalFlash(), PLATFORM_UninitExternalFlash(), PLATFORM_ReinitExternalFlash()
  - Support XTAL32K removal functionatity, use FRO32K instead by setting gBoardUseFro32k_d to 1 in board.h file
  - Support localization boards KW45B41Z-LOC Rev C
  - Low power improvement: New BOARD_InitPins() and BOARD_InitPinButtonBootConfig() called from hardware_init.c
  - Removed KW45_A0_SUPPORT support (dcdc)

  - Bug fixes:
    * Fixed glitches on the serial manager RX when exiting from power down
    * Fixed ADC not deinitialized in clock gated modes in BOARD_EnterLowPowerCb()
    * Fixed UART output flush when going to low power: BOARD_UninitAppConsole()
- [platform]
  - PLATFORM_InitBle(), PLATFORM_SendHci() can now block with timeout if NBU does not answer. Application can register callback function to be notified when it occurs: PLATFORM_RegisterBleErrorCallback()
  - Added API to set and get 32Khz XTAL capacitance values: PLATFORM_GetOscCap32KValue() and PLATFORM_SetOscCap32KValue()
  - Added new Service FWK call gFwkSrvNbuMemFullIndication_c to get NBU mem full indication, register with PLATFORM_RegisterNbuMemErrorCallback()
  - Added support negative value in platform intercore service
- [linker script]
  - Realigned gcc linker script with IAR linker script.
  - Added possibility to redefine cstack_start position
  - Added Possibility to change gNvmSectors in gcc linker script
  - Added dedicated reserved Section in shared memory for LL debugging
- [FreeRTOSConfig.h]
  - Removed unused MACRO configFRTOS_MEMORY_SCHEME and configTOTAL_HEAP_SIZE
- [HW Param]
  - Added xtalCap32K field to store XTAL32K triming value
- [fwk_hal_macros.h]
  - Added MACRO for KB, MB and set, clear bits in bit fields
- [Debug]
  - Added MACROs for performance measurement using DWT: DBG_PERF_MEAS

6.1.3 KW45 MR1 QP1

- [Initialization] Delay the switch to XTAL32K source clock until the BLE host stack is initialized
- [lowpower] NBU wakeup from lowpower: configuration can now be programmed with BOARD_NBU_WAKEUP_DELAY_LPO_CYCLE, BOARD_RADIO_DOMAIN_WAKE_UP_DELAY in board.h file
- [NBU firmware] Major fix for NBU system clock accuracy
- [clock_config]
  - Update SRAM margin and flash config when switching system frequency
  - Trim FIRC in HSRUN case
- [XTAL 32K trim] XTAL 32K configuration can be tuned in board.h file with BOARD_32MHZ_XTAL_TRIM_DEFAULT, BOARD_32KHZ_XTAL_CLOAD_DEFAULT, BOARD_32KHZ_XTAL_COARSE_ADJ_DEFAULT
- [MAC address] Add OUI field in PLATFORM_GenerateNewBDAddr() when using Unique Device Id

6.1.2: RW610/RW612 PRC1

- [Low Power]
  - Updates after SDK Power Manager files renaming.
  - Moved PWR LowPower callback to PLATFORM layers.
  - Bug fixes:
    * Fixed wrong compensation of SysTicks during tickless idle.
    * Reinit RTC bus clock after exit from PM3 (power down).
- [OTA]
  - Initial support for OTA using the external flash.
- [platform]
  - Implemented platform specific time stamp APIs over OSTIMER.
  - Implemented platform specific APIs for OTA and external flash support.
  - Removed PLATFORM_GetLowpowerMode API.
  - Added support of CPU2 wake up over Spinel for OpenThread stack.
  - Bug fixes:
    * Fixed issues related to handling CPU2 power state.
- [board]
  - Updated flash_config to support 64MB range.
- [linker script]
  - Fixed wrong assert.

6.1.1: KW45/K32W1 MR1

- [platform] Use new FLib_MemSet32Aligned() to write in ECC RAM bank to force ECC calculation in the MEM_ReinitRamBank() function
- [FunctionLib] Implement new API to set a word aligned
- [platform] Set coarse amplifier gain of the oscilattor 32k to 3
- [platform] Switch back to RNG for MAC Adress generation
- [SecLib] Get rid of the lowpower constraint of deep sleep in ECDH API
- [DCDC] Set DCDC output voltage to 1.35V in case LDO core is set to 1.1V to ensure a drop of 250mV between them
- [NVM] NvIdle() is now returning the number of operations that has been executed
- [documentation] Add markdown of each framework module by default on all package
- [LowPower] Add a delay advised by hardware team on exit of lowpower for SPC
- [SecLib] Rework of SecLib_mbedTLS ECDH functions
- [OTA] Make OTA_IsTransactionPending() public API
- [FunctionLib] Change prototype of FLib_MemCpyWord(), pDst is now a void* to permit more flexibility
- [NVM] Add an API to know if there is a pending operation in the queue
- [FSCI] Fix wrong error case handling in FSCI_Monitor()

6.1.0: KW45/K32W1 RFP

- [LowPower] Do not call PLATFORM_StopWakeUpTimer() in PWR_EnterLowPower() if PLATFORM_StartWakeUpTimer() was not previously called
- [boards] Add the possibility to wakeup on UART 0 even if it is not the default UART
- [boards] Add support for Hardware flow control for UART0, Enable with gBoardUseUart0HwFlowControl, Pin mux update with two additional API for RTS, CTS pins
- [Sensors] Improve ADC wakeup time from deep sleep state: use save and restore API for ADC context before/after deep sleep state.
- [linker script] update SMU2 shared memory region layout with NBU: increase sqram_btblebuf_size to support 24 connections. Shared memory region moved to the end
- [SecLib] SecLib_DeriveBluetoothSKD() API update to support if EdgeLock key shall be regenerated

6.0.11: KW45/K32W1 PRC3.1

FSCI: Framework Serial Communication Interface

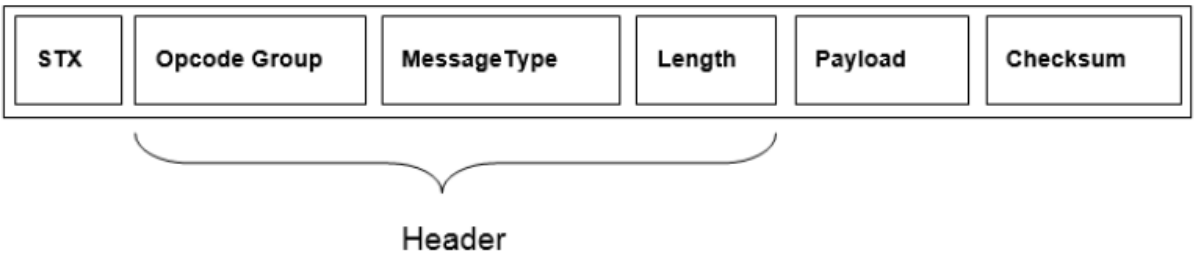
Overview The Framework Serial Communication Interface (FSCI) is both a software module and a protocol that allows monitoring and extensive testing of the protocol layers. It also allows separation of the protocol stack between two protocol layers in a two processing entities setup, the host processor (typically running the upper layers of a protocol stack) and the Black Box application (typically containing the lower layers of the stack, serving as a modem). The Test Tool software is an example of a host processor, which can interact with FSCI Black Boxes at various layers. In this setup, the user can run numerous commands to test the Black Box application services and interfaces.

The FSCI enables common service features for each device enables monitoring of specific interfaces and API calls. Additionally, the FSCI injects or calls specific events and commands into the interfaces between layers.

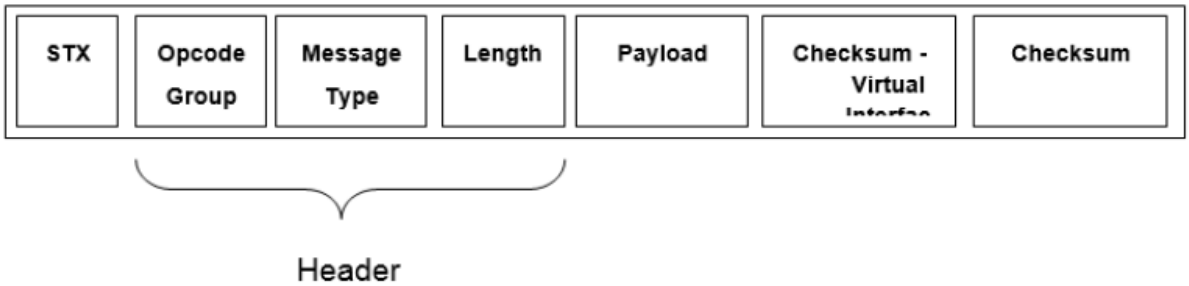
An entity which needs to be interfaced to the FSCI module can use the API to register opcodes to specific interfaces. After doing so, any packet coming from that interface with the same opcode triggers a callback execution. Two or more entities cannot register the same opcode on the same interface, but they can do so on different interfaces. For example, two MAC instances can register the same opcodes, one over UARTA, and the other over UARTB. This way, Test Tool can communicate with each MAC layer over two UART interfaces.

The FSCI module executes either in the context of the Serial Manager task or owns its dedicated task if the compilation Macro *gFsciUseDedicatedTask_c* is set to 1.

FSCI packet structure The FSCI module sends and receives messages as shown in the figure below. This structure is not specific to a serial interface and is designed to offer the best communication reliability. The Black Box device expects messages in little-endian format. It also responds with messages in little-endian format.



Below is an illustration of the FSCI packet structure when a virtual interface is used instead :



Field name	Length (bytes)	Description
STX	1	Used for synchronization over the serial interface. The value is always 0x02.
Opcode Group	1	Distinguishes between different Service Access Primitives (for example MLME or MCPS).
Message Type	1	Specifies the exact message opcode that is contained in the packet.
Length	1 or 2	The length of the packet payload, excluding the header and FCS. The length field content must be provided in little-endian format.
Payload	variable	Payload of the actual message.
Checksum	1	Checksum field used to check the data integrity of the packet.
Checksum2	0 or 1	The second CRC field appears only for virtual interfaces.

NOTE : When virtual interfaces are used, the first checksum is decremented with the ID of the interface. The second checksum is used for error detection.

constant definition The following Macro configurs the FSCI module

```
#define gFsciIncluded_c 0 /* Enable/Disable FSCI module */
#define gFsciUseDedicatedTask_c 1 /* Enable Fsci task to avoid recursivity in Fsci module (Misra␣
↪compliant) */
#define gFsciMaxOpGroups_c 8
#define gFsciMaxInterfaces_c 1
#define gFsciMaxVirtualInterfaces_c 0
#define gFsciMaxPayloadLen_c 245 /* bytes */
#define gFsciTimestampSize_c 0 /* bytes */
#define gFsciLenHas2Bytes_c 0 /* boolean */
#define gFsciUseEscapeSeq_c 0 /* boolean */
#define gFsciUseFmtLog_c 0 /* boolean */
#define gFsciUseFileDataLog_c 0 /* boolean */
#define gFsciLoggingInterface_c 1 /* [0..gFsciMaxInterfaces_c) */
#define gFsciHostMacSupport_c 0 /* Host support at MAC layer */
```

The following provides the OpGroups values reserved by MAC, application, and FSCI.

FSCI Host FSCI Host is a functionality that allows separation at a certain stack layer between two entities, usually two boards running separate layers of a stack.

Support is provided for functionality at the MAC layer, for example, MAC/PHY layers of a stack are running as a Black Box on a board, and MAC higher layers are running on another. The higher layers send and receive serial commands to and from the MAC Black Box using the FSCI set of operation codes and groups.

The protocol of communication between the two is the same. The current level of support is provided for:

- FSCI_MsgResetCPUReqFunc – sends a CPU reset request to black box
- FSCI_MsgWriteExtendedAdrReqFunc – configures MAC extended address to the Black Box
- FSCI_MsgReadExtendedAdrReqFunc – N/A

The approach on the Host interfacing a Black Box using synchronous primitives is by default the polling of the FSCI_receivePacket function, until the response is received from the Black Box. The calling task polls whenever the task is being scheduled. This is required because a stack synchronous primitive requires that the response of that request is available in the context of the caller right after the SAP call has been executed.

The other option, available for RTOS environments, is using an event mechanism. The calling task blocks waiting for the event that is sent from the Serial Manager task when the response is available from the Black Box. This option is disabled by default. The disadvantage of this option is that the primitive cannot be received from another Black Box through a serial interface because the blocked task is the Serial Manager task, which reaches a deadlock as cannot be released again.

FSCI ACK ACK transmission is enabled through the gFsciTxAck_c macro definition. Each FSCI valid packet received triggers an FSCI ACK packet transmission on the same FSCI interface that the packet was received on. The serial write call is performed synchronously to send the ACK packet before any other FSCI packet. Only then the registered handler is called to process the received packet. The ACK is represented by the gFSCI_CnfOpcodeGroup_c and mFsciMsgAck_c Opcode. An additional byte is left empty in the payload so that it can be used optionally as a packet identifier to correlate packets and ACKs. ACK reception is the other component that is enabled through gFsciRxAck_c. The behavior is such that every FSCI packet sent through a serial interface triggers an FSCI ACK packet reception on the same interface after the packet is sent. If an ACK packet is received, the transmission is considered successful. Otherwise, the packet is resent a number of times. The ACK wait period is configurable through mFsciRxAckTimeoutMs_c and the number of transmission retries through mFsciTxRetryCnt_c. The ACK mechanism described above can also be coupled with a FSCI packet reception timeout enabled through gFsciRxTimeout_c and configurable through mFsciRxRestartTimeoutMs_c. Whenever there are no more bytes to be read from a serial interface, a timeout is configured at the predefined value if no other bytes are received. If new bytes are received, the timer is stopped and eventually canceled at successful reception. However, if, for any reason, the timeout is triggered, the FSCI module considers that the current packet is invalid, drops it, and searches for a new start marker.

FSCI usage example Detailed data types and APIs are described in ConnFWK API documentation.

Initialization

```
/* Configure the number of interfaces and virtual interfaces used */
#define gFsciMaxInterfaces_c 4
#define gFsciMaxVirtualInterfaces_c 2
....
/* Define the interfaces used */
```

(continues on next page)

(continued from previous page)

```
static const gFsciSerialConfig_t myFsciSerials[] = {
    /* Baudrate, interface type, channel No, virtual interface */ {gUARTBaudRate115200_c, gSerialMgrUart_
↪c, 1, 0}, {gUARTBaudRate115200_c, gSerialMgrUart_c, 1, 1}, {0 , gSerialMgrIICSlave_c, 1, 0}, {0 ,␣
↪gSerialMgrUSB_c, 0, 0},
};
….
/* Call init function to open all interfaces */
FSCI_Init( (void*)mFsciSerials );
```

Registering operation groups

```
myOpGroup = 0x12; // Operation Group used
myParam = NULL; // pointer to a parameter to be passed to the handler function (myHandlerFunc)
myInterface = 1; // index of entry from myFsciSerials
…
FSCI_RegisterOpGroup( myOpGroup, gFsciMonitorMode_c, myHandlerFunc, myParam, myInterface );
```

Implementing handler function

```
void fsciMcpsReqHandler(void *pData, void* param, uint32_t interfaceId)
{
    clientPacket_t *pClientPacket = ((clientPacket_t*)pData);
    fsciLen_t myNewLen;
    switch( pClientPacket->structured.header.opCode )
    {
        case 0x01:
        {
            /* Reuse packet received over the serial interface The OpCode remains the same. The length of the␣
↪response must be <= that the length of the received packet */
            pClientPacket->structured.header.opGroup = myResponseOpGroup;/* Process packet */
            …
            pClientPacket->structured.header. len = myNewLen;
            FSCI_transmitFormatedPacket(pClientPacket, interfaceId);
            return;
        }
        case 0x02:
        {
            /* Alocate a new message for the response. The received packet is Freed */
            clientPacket_t *pResponsePkt = MEM_BufferAlloc( sizeof(clientPacketHdr_t) + myPayloadSize_d␣
↪+ sizeof(uint8_t) // CRC);
            if(pResponsePkt)
            {
                /* Process received data and fill the response packet */ …
                pResponsePkt->structured.header. len = myPayloadSize_d;
                FSCI_transmitFormatedPacket(pClientPacket, interfaceId);
            }
            break;
        }
        default:
            MEM_BufferFree( pData );
            FSCI_Error( gFsciUnknownOpcode_c, interfaceId );
            return;
    }
    /* Free message received over the serial interface */
    MEM_BufferFree( pData );
}
```

Helper Functions Library

Overview This framework provides a collection of features commonly used in embedded software centered on memory manipulation.

HWParameter: Hardware parameter

Production Data Storage Hardware parameters provide production data storage

Overview Different platforms/boards need board/network node-specific settings to function according to the design. (Examples of such settings are IEEE® addresses and radio calibration values specific to the node.) For this purpose, the last flash sector is reserved and contains hardware-specific parameters for production data storage. These parameters pertain to the network node as a distinct entity. For example, a silicon mounted on a PCB in a specific configuration, rather than to just the silicon itself. This sector is reserved by the linker file, through the PROD_DATA section and it should be read/written only through the API described below.

> Note : This sector is not erased/written at code download time and it is not updated via over-the-air firmware update procedures to preserve the respective node-specific data, regardless of the firmware running on it.

Constant Definitions Name :

```
extern uint32_t PROD_DATA_BASE_ADDR[];
```

Description :

This symbol is defined in the linker script. It specifies the start address of the PROD_DATA section.

Name :

```
static const uint8_t mProdDataIdentifier[10] = {"PROD_DATA:"};
```

Description :

The value of this constant is copied as identification word (header) at the beginning of the PROD_DATA area and verified by the dedicated read function.

Note: the length of mProdDataIdentifier imposes the definition of PROD_DATA_ID_STRING_SZ as 10. The legacy HW parameters structure provides headroom for future usage. There are currently 63 bytes available.

Data type definitions Name :

```
typedef PACKED_STRUCT HwParameters_tag
{
    uint8_t identificationWord[PROD_DATA_ID_STRING_SZ]; /* internal usage only: valid data present */
    /*@{*/
    uint8_t bluetooth_address[BLE_MAC_ADDR_SZ];       /*!< Bluetooth address */
    uint8_t ieee_802_15_4_address[IEEE_802_15_4_SZ]; /*!< IEEE 802.15.4 MAC address - K32W1 only␣
↪*/
    uint8_t xtalTrim;                                 /*!< XTAL 32MHz Trim value */
    uint8_t xtalCap32K;                               /*!< XTAL 32kHz capacitance value */
    /* For forward compatibility additional fields may be added here
       Existing data in flash will not be compatible after modifying the hardwareParameters_t typedef.
       In this case the size of the padding has to be adjusted.
    */
    uint8_t reserved[1];
    /* first byte of padding : actual size if 63 for legacy HwParameters but
```

(continues on next page)

(continued from previous page)

```
    complement to 128 bytes in the new structure */
}
hardwareParameters_t;
```

Description:

Defines the structure of the hardware-dependent information.

> Note : Some members of this structure may be ignored on a specific board/silicon configuration. Also, new members may be added for implementation-specific purposes and the backward compatibility must be maintained.

The CRC calculation starts from the reserved field of the hardwareParameters_t and ends before the hardwareParamsCrc field. Additional members to this structure may be added using the following method :

Add new fields before the reserved field. This method does not cause a CRC fail, but you must keep in mind to subtract the total size of the new fields from the size of the reserved field. For example, if a field of uint8_t size is added using this method, the size of the reserved field shall be changed to 63.

Co-locating application factory data in HW Parameters flash sector. The sector containing the Hardware parameter structure may be located in the internal flash, usually at its last sector. The actual Hardware parameter structure has a size of 128 bytes - including padding reserved for future use. Since there is plenty of room available in a flash sector (4kB or 8kB), co-locating Application Factory Data in the same structure prevents from reserving another flash sector for these data. The application designer may adopt this solution by defining gHwParamsAppFactoryDataExtension_d as 1. A total of 2kB is alloted to this purpose.

If this option was chosen, whenever any of the Hardware parameter fields is modified, its CRC16 will change so the sector will need erasing. The gHwParamsAppFactoryDataPreserveOnHwParamUpdate_d compilation option deals with restoring the contents of the App Factory Data. Nonetheless this requires a temporary allocation a 2kB buffer to preserve the previous content and restore then on completion of the Hw Parameter update.

Special reserved area at start of IFR1 in range [0x02002000..0x02002600[On development boards a 1536 byte area is reserved and the actual Hardware parameter area begins at offset 0x600. Preserving this area on a HW parameter update also requires a temporary 1.5kB dynamic allocation (in addition to the App Factory 2kB allocation), to be able to restore on completion of update operation.

HW Parameters Production Data placement options The placement of production data (PROD_DATA) can be selected based on the definition of gHwParamsProdDataPlacement_c (see fwk_config.h). The productions data seldom need update for final products, once calibration data, MAC addresses or others have been programmed. Two cases exist, plus a transition mode :

1) gHwParamsProdDataMainFlashMode_c (0) :
   - PROD_DATA are located at top of Main Flash. Hardware parameters section is placed in the last sector of internal flash [0xfe000..0x100000[.
   - The linker script must reserve this area explicitly so as to prevent placement of NVM or text sections at that location by setting gUseProdInfoMainFlash_d.
2) gHwParamsProdDataMainFlash2IfrMode_c(1) : - PROD_DATA are located in IFR1, but MainFlash version still exists during interim period. - If the contents of the PROD_DATA section in MainFlash is valid (not blank and correct CRC) but the IFR PROD_DATA is still blank, copy the contents of MainFlash PROD_DATA to IFR location. - When done PROD_DATA in IFR are

used. Once the transition is done, an application using (2: gHwParamsProdDataPlacementIfrMode_c) may be programmed.

3) gHwParamsProdDataIfrMode_c (2) :
   - PROD_DATA section dwells in the IFR1 sector [0x02002000..0x02004000[
   - in development phase the area comprised between [0x02002000..0x02002600[must be reserved for internal purposes.
   - This allows to free up the top sector of Main Flash by linking with gUseProdInfoMainFlash_d unset.

LowPower

Low Power reference user guide This Readme file describes the connectivity software architecture and provides the general low power enablement user guide.

1- Connectivity Low Power SW architecture The connectivity low power software architecture is composed of various components. These are described from the lower layer to the application layer:

1. The SDK power manager in component/power_manager. This component provides the basic low power framework. It is not specific to the connectivity but generic across devices. it covers:
   - gather the low power constraints for upper layer and take the decision on the best suitable low power state the device is allowed to go to fullfill the constraints.
   - call the low power entry and exit function callbacks
   - call the appropriate SW routines to switch the device into the suitable low power state
2. Connectivity Low power module in the connectivity framework. This module is composed of:
   - The low power service called PWR inside framework/LowPower (this folder), This module is generic to all connectivity devices.
   - The platform lowpower: fwk_platform_lowpower.[ch] located in framework\platform\<platform_name>. These files are a collection of low power routines functions for the PWR module and upper layer. These are specific to the device.

     Both PWR and platform lowpower files are detailed in section below.
3. Low power Application modules, it consists of 3 parts:
   - Application initialization file app_services_init.c where the application initializes the low power framework, see next section 'Demo example for typical usage of low power framework'
   - Application Idle task from application to call the main low power entry function PWR_EnterLowPower() to switch the device into lowpower. This function is application specific, one example is given in the section 1.3.3
   - Low power board files : board_lp.[ch] located in board/lowpower. These files implement the low power entry and exit functions related to the application and board. Customers shall modify these files for their own needs. Example code is given for the connectivity applications.

     User guide is provided in section 1.3 below.

> **Note :** Linker script may also be impacted for power down mode support in order to provide an RAM area for ROM warm boot (depends on the platform) and application warmboot stack

The Low power central and master reference design applications provide an example of Low power implementation for BLE. Customer can also refer to the associated document 'low power connectivity reference design user guide'.

1.1 - SDK power manager This module provides the main low power functionalities such as:

- Decide the best low-power mode dependent on the constraints set by upper layers by using PWR_SetLowPowerModeConstraints() API function.
- Handle the sequences to enter and exit low-power mode.
- Enable and configure wake up sources, call the application callbacks on low power entry/exit sequences.

The SDK power manager provides the capability for application and all components to receive low power constraints to the power. The Application does not set the low-power mode the device shall go into. When going to low power, the SDK power manager selects the best low-power mode that fits all the constraints.

As an example, if the low power constraint set from Application is Power Down mode, and no other constraint is set, the SDK power manager selects Power down mode, the next time the device enters low power. However, if a new constraint is set by another component, such as the SecLib module that operates Hardware encryption, the SecLib module would select WFI as additional low power constraint. Also, the SDK power manager selects this last low-power mode until the constraint is released by the SecLib module. It then reselects Power Down mode for further low power entry modes.

1.2 - PWR Low power module The PWR module in the connectivity framework provides additional services for the connectivity stacks and applications on top of the SDK power manager.

It also provides a simple API for Connectivity Stack and Connectivity applications.

However, more advanced features such as configuring the wake-up sources are only accessible from the SDK Power Manager API.

In addition to the SDK Power Manager, the PWR module uses the software resources from lower level drivers but is independent of the platform used.

1.2.1 - Functional description Initialization of the PWR module should be done through PWR_Init() function. This is mainly to initialize the SDK power manager and the platform for low power. It also registers PWR low power entry/exit callback PWR_LowpowerCb() to the SDK power manager. This function will be called back when entering and exiting low power to perform mandatory save/restore operations for connectivity stacks. The application can perform extra optional save/restore operations in the board_lp file where it can register to the SDK Power Manager its own callback. This is usually used to handle optional peripherals such as serial interfaces, GPIOs, and so on.The main entry function is PWR_EnterLowPower(). It should be called from Idle task when no SW activity is required. The maximum duration for lowpower is given as argument timeoutUs in useconds. This function will check the next Hardware event in the connectivity stack, typically the next Radio activity. A wakeup timer is programmed if the timeoutUs value is shorter than the next radio event timing. Passing a timeout of 0us will be interpreted as no timeout on the application side.

On device wakeup from low power state, the function will return the time duration the device has been in low power state.

Two APi are provided to set and release low power state constraints : PWR_SetLowPowerModeConstraint() and PWR_ReleaseLowPowerModeConstraint(). These are helper functions. User can use directly the SDK power manager if needed.

The PWR module also provides some API to be set as callbacks into other components to prevent from going to low power state. It can be used in following examples :

1. If a DMA is running, the module in charge of the DMA would need to set a constraint to avoid the system from going to a low power state when the RAM and system bus are no longer available.
2. If transfer is going on a peripheral, the drivers shall set a constraint to forbid low power mode.
3. If encryption is on going through an Hardware accelerator, the HW accelerator and the required ressources (clocks, etc), shall be kept active also by setting a constraints.

1.2.2 - Tickless mode support This module also provides some routines functions PWR_SysticksPreProcess() and PWR_SysticksPostProcess() from PWR_systicks.c in order to support the tickless mode when using FreeRTOS. The tickless mode is the capability to suspend the periodic system ticks from FreeRTOS and keep timebase tracking using another low power counter. In this implementation, the Timer Manager and time_stamp component are used for this purpose.

Idle task shall call these functions PWR_SysticksPreProcess() and PWR_SysticksPostProcess() before and after the call to the main low power entry function PWR_EnterLowPower().

Refer to framework/LowPower/PWR_systicks.c file or section 2.1 below for more information.

1.3 - Low power platform submodule Low power platform module file fwk_platform_lowpower.c provides the necessary helper functions to support low power device initialization, device entry, and exit routines. These are platform and device specific. Typically, the PWR module uses the low power platform submodule for all low power specific routines.

The low power platform submodule is documented in the Connectivity Framework Reference Manual document and in the Connectivity Framework API document.

1.4 - Low power board files Low power board files board_lp.[ch] are both application and board specific. Users should update this file to add new functions to include new used peripherals that require low power support. In the current SDK package, only Serial Manager over UART and button (IO toggle wake up source) are supported and demonstrated in the Bluetooth LE demo application.

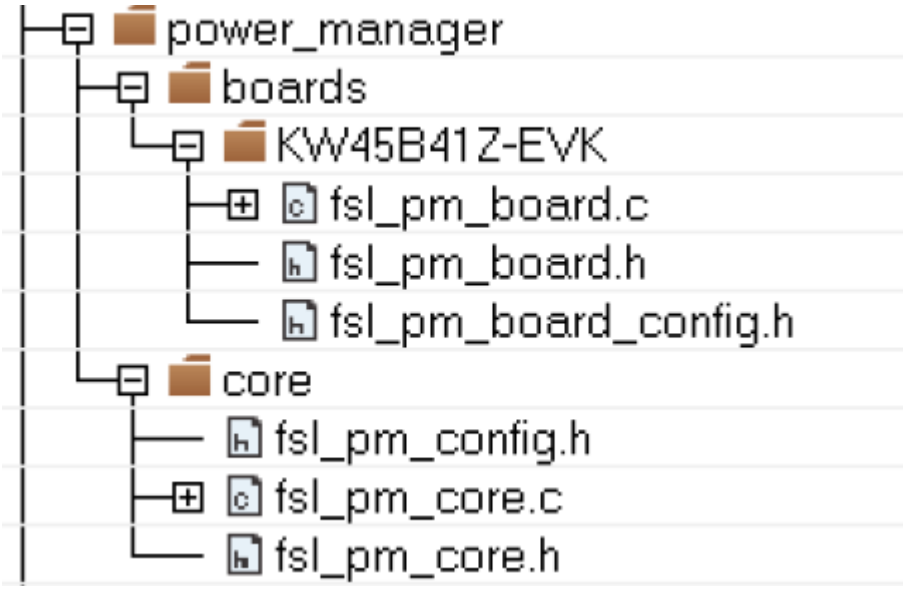
Other peripherals that require specific action on low power entry and restore on low power exit should be added to low power board files. For more details, refer to section Low power board file update

2 - Low power Application user guide This section provides a user guide to enable Low power on a connectivity application, It gives example of typical implementation for the initialization, Idle task function and low power entry/exit functions.

2.1 - Application Project updates It is recommended to reuse the low-power peripheral/central reference design application projects as a start. This ensures that everything is in place for the low-power optimization feature. Then, application files may be added to one of the two projects.

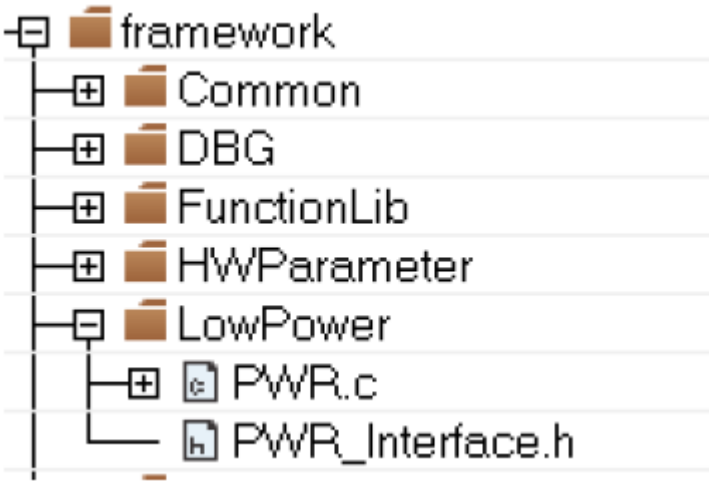
However, users can start directly from the application project and implement low power in it, by performing the steps described in the following sections.

2.1.1 - SDK Power Manager Most of the Low power functionality is implemented in the SDK Power Manager. The files to add into the project SDK power_manager module are listed in the figure below:



You need to use the files located in the folder that match your device.

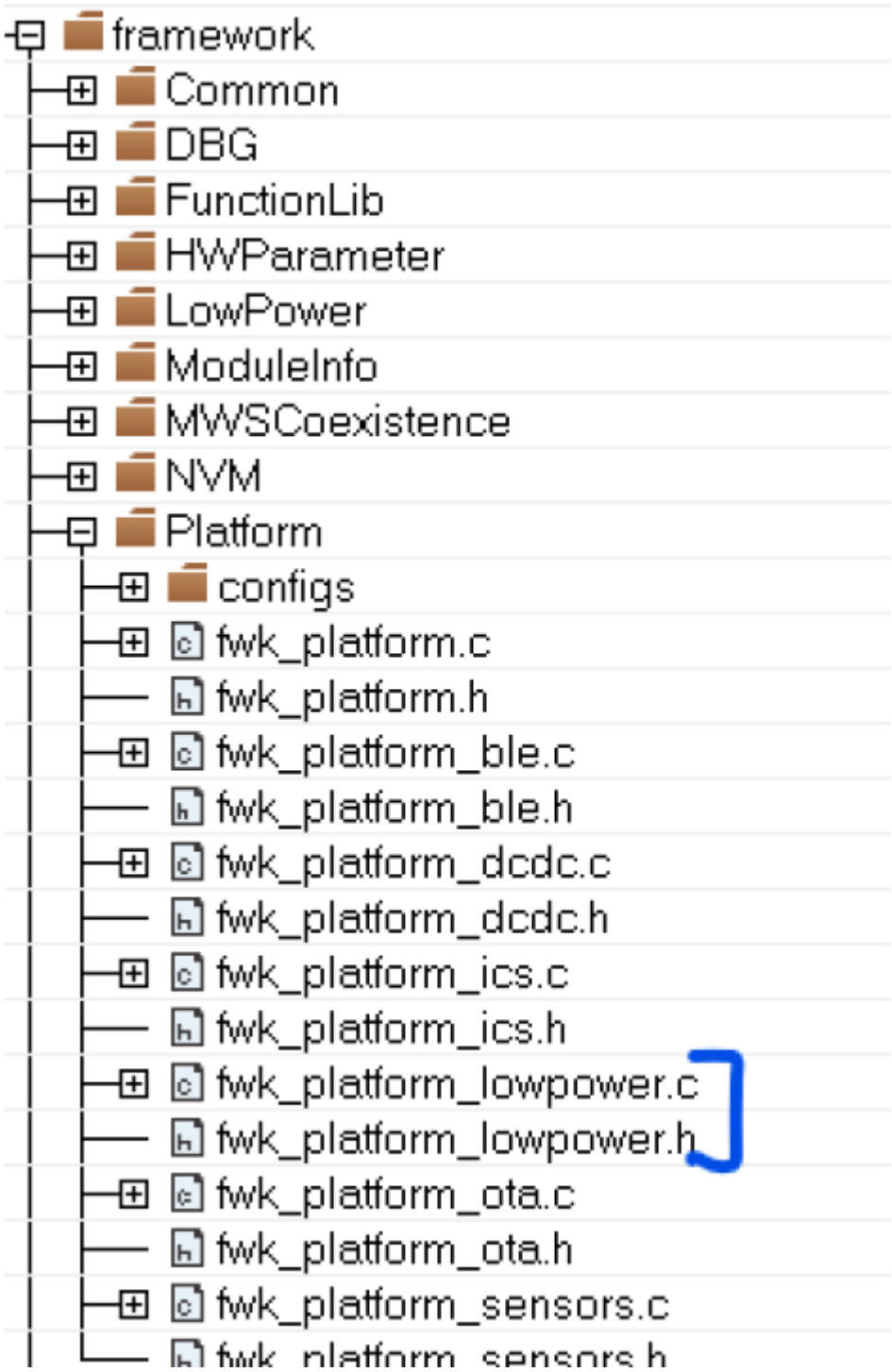
2.1.2 - PWR connectivity framework module PWR.c PWR_Interface.h shall be added to your application projects :



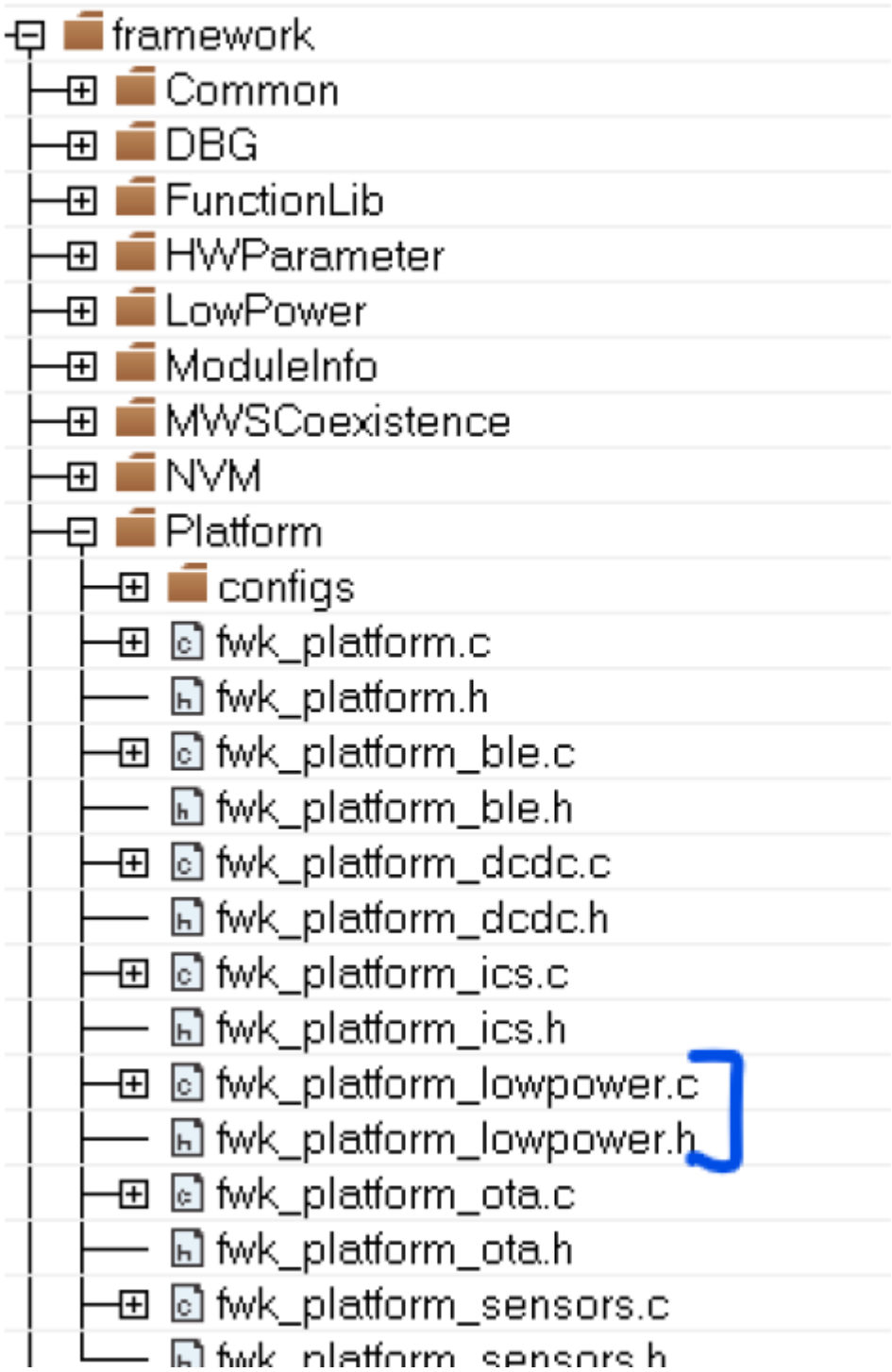
Optionally, in order to support Systick less mode, PWR_systicks.c or PWR_systicks_bm.c could also be added.

The include path to add is: middleware/wireless/framework/LowPower

2.1.3 -Low power platform submodule Low power platform files can be found in the 'Platform' module in the connectivity framework:



2.1.4 - Low power board files These files are located in the same folder that the other board files board.[ch]. Hence, it is not required to add any new include path at compiler command line.



2.1.5 - Application RTOS Idle hook and tickeless hook functions See section 2.4.3 Idle task implementation example

2.2 - Low power and wake up sources Initialization Low power initialization and configuration are performed in APP_ServiceInitLowpower()function. This is called from APP_InitServices() function called from the main() function so all is already set up when calling the main application entry point, typically BluetoothLEHost_AppInit() function in the Bluetooth LE demo applications.

The default Low Power mode configured in APP_InitServices() is Deep Sleep mode. In Bluetooth

LE, (or any other stack technology), Deep Sleep mode fits for all use cases. For instance, for Bluetooth LE states: Advertising, Connected, Scanning states. This mode already performs a very good level of power saving and likely, this is not required to optimize more if the device is powered from external supply.

APP_ServiceInitLowpower() function performs the following initialization and configuration:

- Initialize the Connectivity framework Low power module PWR_Init(), this function initialized the SDK power manager.
- Configure the wakeup sources such as serial manager wake up source for UART, or button for IO wake up configuration. These are typical wakeup sources used in the connectivity application. Developer may want to add additional wake up sources here specific for the application.

  Note : The low power timer wakeup source and wakeup from Radio domain are directly enabled from the Connectivity framework Low power module PWR as it is mandatory for the connectivity stack. If your application supports other peripherals (such as i2c, spi, and others) that require wake sources from low power, developer should add additional wake up sources setting in this function APP_ServiceInitLowpower(). The complete list of wakeup sources are available from the SDK power manager component, see file fsl_pm_board.h in component/boards/<device_name>/.
- Initialize and register the Low power board file used to register and implement low power entry and exit callback function used for peripheral. This is done by calling the BOARD_LowPowerInit() function.
- Register low power Enter and exit critical function to driver component to enable / disable low power when the Hardware is active. Example is given for serial manager that needs to disable low power when the TX ring buffer contains data so the device does not enter low power until the buffer is empty.

Finally, APP_ServiceInitLowpower() function configures the Deep Sleep mode as the default low power constraint for the application. It is recommended to keep this level of low power constraint during all the connectivity stack initialization.

Example of low power framework initialization can be found in app_services_init.c file. Below is some code example for initializing the low power framework and wake up sources:

```
static void APP_ServiceInitLowpower(void)
{
    PWR_ReturnStatus_t status = PWR_Success;

    /* It is required to initialize PWR module so the application
     * can call PWR API during its init (wake up sources...) */
    PWR_Init();

    /* Initialize board_lp module, likely to register the enter/exit
     * low power callback to Power Manager */
    BOARD_LowPowerInit();

    /* Set Deep Sleep constraint by default (works for All application)
     *   Application will be allowed to release the Deep Sleep constraint
     *   and set a deepest lowpower mode constraint such as Power down if it needs
     *   more optimization */
    status = PWR_SetLowPowerModeConstraint(PWR_DeepSleep);
    assert(status == PWR_Success);

#if (defined(gAppButtonCnt_c) && (gAppButtonCnt_c > 0))

    /* Init and enable button0 as wake up source
     * BOARD_WAKEUP_SOURCE_BUTTON0 can be customized based on board configuration
```

(continues on next page)

(continued from previous page)

```
    * On EVK we use the SW2 mapped to GPIOD */
   PM_InitWakeupSource(&button0WakeUpSource, BOARD_WAKEUP_SOURCE_BUTTON0, NULL,␣
↪true);
#endif

#if (gAppButtonCnt_c > 1)
   /* Init and enable button1 as wake up source
    * BOARD_WAKEUP_SOURCE_BUTTON1 can be customized based on board configuration
    * On EVK we use the SW3 mapped to PTC6 */
   PM_InitWakeupSource(&button1WakeUpSource, BOARD_WAKEUP_SOURCE_BUTTON1, NULL,␣
↪true);
#endif

#if (defined(gAppUseSerialManager_c) && (gAppUseSerialManager_c > 0))

#if defined(gAppLpuart0WakeUpSourceEnable_d) && (gAppLpuart0WakeUpSourceEnable_d > 0)
   /* To be able to wake up from LPUART0, we need to keep the FRO6M running
    * also, we need to keep the WAKE domain is SLEEP.
    * We can't put the WAKE domain in DEEP SLEEP because the LPUART0 is not mapped
    * to the WUU as wake up source */
   (void)PM_SetConstraints(PM_LP_STATE_NO_CONSTRAINT, APP_LPUART0_WAKEUP_
↪CONSTRAINTS);
#endif

   /* Register PWR functions into SerialManager module in order to disable device lowpower
      during SerialManager processing. Typically, allow only WFI instruction when
      uart data are processed by serail manager  */
   SerialManager_SetLowpowerCriticalCb(&gSerMgr_LowpowerCriticalCBs);
#endif

#if defined(gAppUseSensors_d) && (gAppUseSensors_d > 0)
   Sensors_SetLowpowerCriticalCb(&app_LowpowerSensorsCriticalCBs);
#endif

   (void)status;
}
```

2.3 - low power entry/exit sequences : board files updates Board Files that handles low-power are board_lp.[ch] files.

Low power board files implement the low-power callbacks of the peripherals to be notified when entering or exiting Low Power mode. This module also registers these low-power callbacks to the SDK Power Manager component to get the notifications when the device is about to enter low-power or exit Low Power mode. The Low-power callbacks are registered from BOARD_LowPowerInit() function. This function is called from app_services_init.c file after PWR module initialization.

The low power callback functions can be categorized in two groups:

- Entry Low power call back functions: These are usually used to prepare the peripherals to enter low-power. For example, they can be used for flushing FIFOs, switching off some clocks, and reconfiguring pin mux to avoid leakage on pins. In case of Power Down mode, these functions could be used to save the Hardware peripheral context.
- Exit Low power call back functions: These are typically used to restore the peripherals to functionality. Therefore, they perform the reverse of what is done by the entry callback functions: restoring the pin mux, re-enabling the clock, in case of Power Down mode, restoring the Hardware peripheral context, and so on.

Note that distinction can be done between clock gating mode (Deep Sleep mode), and power gated mode (Power down mode) when entering and exiting Low Power mode. The

BOARD_EnterLowPowerCb() and BOARD_ExitLowPowerCb() functions provide the code to call the various peripheral entry and exit functions to go and exit Deep Sleep mode: serial manager, button, debug console, and others.

However, the processing to save and restore the Hardware peripheral is implemented in different functions BOARD_EnterPowerDownCb() and BOARD_ExitPowerDownCb(). These two functions should be called when exiting power gated modes of the power domain. These two should implement specific code for such case (likely the complete reinitialization of each peripheral). In order to know the Low Power mode that the wake up domain, or main domain has been entered, the low-power platform API PLATFORM_GetLowpowerMode() can be called.

> **Note :** BOARD_ExitPowerDownCb() is called before BOARD_ExitLowPowerCb() as it is generally required to restore the Hardware peripheral contexts before reconfiguring the pin mux to avoid any signal glitches on the pads

Also, It is important to know whether the location of the Hardware peripheral is in the main domain or wake up domain. The two power domains can go into different power modes with the limitation that the wakeup domain cannot go to a deepest Low Power mode than the main domain. Depending on the constraint set on SDK power manager, the wake up domain could remain in active while the main domain can go to deep sleep or power down modes. In this case, the peripherals in the wake up domain does not required to be restored, as explained in the section Power Down. Likely, only pin mux reconfiguration is required in this case.

example Low power entry and exit functions shall be registered to the SDK power manager so these functions will be called when the device will enter and exit low power mode. This is done by BOARD_LowPowerInit() typically called from application source code in app_services_init.c file

```
static pm_notify_element_t boardLpNotifyGroup = {
    .notifyCallback = BOARD_LowpowerCb,
    .data           = NULL,
};

void BOARD_LowPowerInit(void)
{
    status_t status;

    status = PM_RegisterNotify(kPM_NotifyGroup2, &boardLpNotifyGroup);
    assert(status == kStatus_Success);
    (void)status;
}
```

BOARD_LowpowerCb() callback function will handle both the entry and exit sequences. An argument is passed to the function to indicate the lowpower state the device enter/exit. Typical implementation is given below. Customer shall make sure to differentiate low power entry and exit, and the various low power states.

Typically, nothing is expected to be done if low power state is WFI or Sleep mode. These modes are some light low power states and the system can be woken up by interrupt trigger.

In Deep sleep mode, the clock tree and source clocks are off, the system needs to be woken up from an event from the WUU module.

In Power down mode, some peripherals are likely to be powered off, context save and restore may need to be done in these functions.

```
static status_t BOARD_LowpowerCb(pm_event_type_t eventType, uint8_t powerState, void *data)
{
    status_t ret = kStatus_Success;
    if (powerState < PLATFORM_DEEP_SLEEP_STATE)
    {
        /* Nothing to do when entering WFI or Sleep low power state
           NVIC fully functionnal to trigger upcoming interrupts */
```

(continues on next page)

(continued from previous page)

```
    }
    else
    {
        if (eventType == kPM_EventEnteringSleep)
        {
            BOARD_EnterLowPowerCb();

            if (powerState >= PLATFORM_POWER_DOWN_STATE)
            {
                /* Power gated low power modes often require extra specific
                 * entry/exit low power procedures, those should be implemented
                 * in the following BOARD API */
                BOARD_EnterPowerDownCb();
            }
        }
        else
        {
            /* Check if Main power domain domain really went to Power down,
             *   powerState variable is just an indication, Lowpower mode could have been skipped by an␣
↪immediate wakeup
             */
            PLATFORM_PowerDomainState_t main_pd_state = PLATFORM_NO_LOWPOWER;
            PLATFORM_status_t           status;

            status = PLATFORM_GetLowpowerMode(PLATFORM_MainDomain, &main_pd_state);
            assert(status == PLATFORM_Successful);
            (void)status;

            if (main_pd_state == PLATFORM_POWER_DOWN_MODE)
            {
                /* Process wake up from power down mode on Main domain
                 *  Note that Wake up domain has not been in power down mode */
                BOARD_ExitPowerDownCb();
            }

            BOARD_ExitLowPowerCb();
        }
    }
    return ret;
}
```

2.4 - Low power constraint updates and optimization Except for the board file update as seen in previous section, the application does not need any other changes for low-power support in Deep Sleep mode. It shall work as if no low-power is supported. However, If more aggressive power saving is required, this constraint can be changed in your application in order to further reduce the power consumption in Low Power mode.

2.4.1 - Changing the Default Application low power constraint after firmware initialization The Low power reference design applications (central or peripheral) provides demonstration on how to change the Application low power constraint. In the Application main entry point BluetoothLEHost_AppInit(), Deep Sleep mode is configured by default from APP_ServiceInitLowpower() function.

> **Note :** It is recommended to keep Deep Sleep mode as default during all the stack initialization phase until BluetoothLEHost_Initialized() and BleApp_StartInit() functions are called. In case of Bonded device with privacy, it is recommended to wait for gControllerPrivacyStateChanged_c event to be called.

BleApp_LowpowerInit() function provides an example of code on how to release the default Deep sleep low-power constraint and set a new constraint such as Power down mode for the application. This deeper low-power mode is used when no Bluetooth LE activity is on going, and if no other higher Low-power constraint is set by another components or layer. For instance, if some serial transmission is on going by the serial manager, or if the SecLib module has on going activity on the HW crypto accelerator, the low-power mode could less deep.

```
static void BleApp_LowpowerInit(void)
{
#if defined(gAppLowpowerEnabled_d) && (gAppLowpowerEnabled_d>0)
    PWR_ReturnStatus_t status;

    /*
     * Optionally, Allow now Deepest lowpower mode constraint given by gAPP_
↪LowPowerConstraintInNoBleActivity_c
     *    rather than DeepSleep mode.
     * Deep Sleep mode constraint has been set in APP_InitServices(), this is fine
     *    to keep this constraint for typical lowpower application but we want the
     *     lowpower reference design application to be more agressive in term of power saving.

     *   To apply a lower lowpower mode than Deep Sleep mode, we need to
     *     - 1) First, release the Deep sleep mode constraint previously set  by default in app_services_init()
     *     - 2) Apply new lowpower constraint when No BLE activity
     *   In the various BLE states (advertising, scanning, connected mode), a new Lowpower
     *     mode constraint will be applied depending of Application Compilation macro set in app_preinclude.
↪h :
     *     gAppPowerDownInAdvertising, gAppPowerDownInConnected, gAppPowerDownInScanning
     */

    /*  1) Release the Deep sleep mode constraint previously set  by default in app_services_init() */
    status = PWR_ReleaseLowPowerModeConstraint(PWR_DeepSleep);
    assert(status == PWR_Success);
    (void)status;

    /*  2) Apply new Lowpower mode constraint gAppLowPowerConstraintInNoBleActivity_c *
     *       The BleAppStart() call above has already set up the new lowpower constraint
     *         when Advertising request has been sent to controller        */
    BleApp_SetLowPowerModeConstraint(gAppLowPowerConstraintInNoBleActivity_c);
#endif
}
```

2.4.2 - Changing the Application lowest low power constraint during application execution

In the various application use cases, (in the various Bluetooth LE activity states, advertising, connected, scanning), some lower low-power constraint can be set, as Power down for advertising, Deep Sleep for connected, or Scanning. Customer can change the level of Low Power mode in the various use case mainly depending of the time duration the device is supposed to remain in low-power. The longer the time that the device remains in low power, the higher the benefit for a deeper Low Power mode such as Power down mode. However, please note that the wake up from power down mode takes significantly more time than deep sleep as ROM code is re executed and the hardware logic needs to be restored. Sections Deep Sleep and Power Down provide some guidance on when to use Deep Sleep mode or Power Down modes respectively.

In the low power reference design applications, four application compilations macros are defined to adjust the low-power mode into advertising, scanning, connected, or no Bluetooth LE activity. Other use cases can be added as desired. For instance, If application needs to run a DMA transfer, or if application needs to wakeup regularly to process data from external device, it may be useful to set WFI constraint (in case of DMA transfer), or Deep Sleep constraint (in case of regular wake up to process external data), rather than power down or a even lower low-power mode.

The 4 application compilation macros can be found in app_preinclude.h file of the project. See

app_preinclude.h for low power reference design peripheral application :

```
/*! Lowpower Constraint setting for various BLE states (Advertising, Scanning, connected mode)
   The value shall map with the type defintion PWR_LowpowerMode_t in PWR_Interface.h
     0 : no LowPower, WFI only
     1 : Reserved
     2 : Deep Sleep
     3 : Power Down
     4 : Deep Power Down
   Note that if a Ble State is configured to Power Down mode, please make sure
      gLowpowerPowerDownEnable_d variable is set to 1 in Linker Script
   The PowerDown mode will allow lowest power consumption but the wakeup time is longer
      and the first 16K in SRAM is reserved to ROM code (this section will be corrupted on
      each power down wakeup so only temporary data could be stored there.)
          Power down feature not supported. */

#define gAppLowPowerConstraintInAdvertising_c          3
/* Scanning not supported on peripheral */
//#define gAppLowPowerConstraintInScanning_c            2
#define gAppLowPowerConstraintInConnected_c            2
#define gAppLowPowerConstraintInNoBleActivity_c        4
```

In lowpower_central.c lowpower_preripheral.c files, the application sets and releases the low power constraint from BleApp_SetLowPowerModeConstraint() and BleApp_ReleaseLowPowerModeConstraint() functions. These functions are called with the macro value passed as argument.

> **Important Note :** Setting the application low power constraint shall be done on new Bluetooth LE state request so the new constraint is applied immediately, while the application low-power mode constraint shall be released when the Bluetooth LE state is exited. For example, setting the new low power constraint for Advertising shall be done when the application requests advertising to start. Releasing the low power constraint shall be done in the advertising stop callback (advertising has been stopped).

After releasing the low power constraint, the previous low power constraint, (likely the one that has been set during firmware initialization in APP_ServiceInitLowpower() function, or the updated low power constraint in BleApp_StartInit() function) applies again.

2.4.3 - Idle task implementation example

2.4.3.1 Tickless mode support and Low power entry function Idle task configuration from FreeRTOS shall be enabled by configUSE_TICKLESS_IDLE in FreeRTOSConfig.h. This will have the effect to have vPortSuppressTicksAndSleep() called from Idle task created by FreeRTOS. Here is a typical implementation of this function:

```
void vPortSuppressTicksAndSleep(TickType_t xExpectedIdleTime)
{

    bool abortIdle = false;
    uint64_t actualIdleTimeUs, expectedIdleTimeUs;

    /* The OSA_InterruptDisable() API will prevent us to wakeup so we use
     * OSA_DisableIRQGlobal() */
    OSA_DisableIRQGlobal();

    /* Disable and prepare systicks for low power */
    abortIdle = PWR_SysticksPreProcess((uint32_t)xExpectedIdleTime, &expectedIdleTimeUs);

    if (abortIdle == false)
    {
```

(continues on next page)

(continued from previous page)

```
        /* Enter low power with a maximal timeout */
        actualIdleTimeUs = PWR_EnterLowPower(expectedIdleTimeUs);

        /* Re enable systicks and compensate systick timebase */
        PWR_SysticksPostProcess(expectedIdleTimeUs, actualIdleTimeUs);
    }

    /* Exit from critical section */
    OSA_EnableIRQGlobal();
}
```

2.4.3.2 Connectivity background tasks and Idle hook function example Some process needs to be run in background before going into low power. This is the case for writing in NVM, or firmware update OTA to be writen in Flash. If so, configUSE_IDLE_HOOK shall be enabled in FreeRTOSCOnfig.h so vApplicationIdleHook() will be called prior to vPortSuppressTicksAndSleep(). Typical implementation of vApplicationIdleHook() function can be found here :

```
void vApplicationIdleHook(void)
{
    /* call some background tasks required by connectivity */
#if ((gAppUseNvm_d) || \
    (defined gAppOtaASyncFlashTransactions_c && (gAppOtaASyncFlashTransactions_c > 0)))

    if (PLATFORM_CheckNextBleConnectivityActivity() == true)
    {
        BluetoothLEHost_ProcessIdleTask();
    }
#endif
}
```

PLATFORM_CheckNextBleConnectivityActivity() function implemented in low power platform file fwk_platform_lowpower.c typically checks the next connectivity event and returns true if there's enough time to perform time consuming tasks such as flash erase/write operations (can be defined by the compile macro depending on the platform).

2. Low power features

2.1 - FreeRTOS systicks Low power module in framework supports the systick generation for FreeRTOS. Systicks in FreeRTOS are most of the time not required in the Bluetooth LE demos applications because the framework already supports timers by the timer manager component, so the application can use the timers from this module. The systicks in FreeRTOS are useful for all internal timer service provided by FreeRTOS (through OSA) like OSA_TimeDelay(), OSA_TimeGetMsec(), OSA_EventWait(). When systicks are enabled, an interrupt (systick interrupt) is triggered and executed on a periodic basis. In order to save power, periodic systick interrupts are undesirable and thus disabled when going to low-power mode. This feature is called low power FreeRTOS tickless mode. When entering the low power state, the system ticks shall be disabled and switch to a low power timer. On wake-up, the module retrieves the time passed in low power and compensate the ticks count accordingly. This feature does not apply on bare metal scheduler.

On FreeRTOS, the vPortSuppressTicksAndSleep() function implemented in the app_low_power.c file will be called when going to idle. FreeRTOS will give to this function the xExpectedIdleTime, time in tick periods before a task is due to be moved into the Ready state. This function will manage the systicks (disable/enable) through PWR_SysticksPreProcess() and PWR_SysticksPostProcess() calls. Then, when calling PWR_EnterLowPower(), a time out duration in micro seconds will be given and the function will set a timer before entering low power.

In addition, this function will return the low power period duration, used to compensate the ticks count.

In our example low power reference design peripheral application, an OSA_EventWait() has been added to demonstrate the tickless mode feature. You can adjust the timeout with the gAppTaskWaitTimeout_ms_c flag in the app_preinclude.h file, its value in our demo is 8000ms. So 8 seconds after stopping any activity we will wake up from low power. If the flag is not defined in the application its value will be osaWaitForever_c and there will be no OS wake up.

2.2 - Selective RAM bank retention To optimize the consumption in low power, the linker script specific function PLATFORM_GetDefaultRamBanksRetained() is implemented. This function obtains the RAM banks that need to be retained when the device goes in low power, in order to set them with PLATFORM_SetRamBanksRetained() function. The RAM banks that are not needed are set in power off state, when the device goes in low power mode.

The function PLATFORM_GetDefaultRamBanksRetained() is linker script specific. Hence, it cannot be adapted for a different application. If these functions are called from board_lp.c, it is possible to give to PLATFORM_SetRamBanksRetained() a different bank_mask adapted to your specific application.

In deep power down, this feature does not have any impact because in this power mode, all RAM banks are already powered off.

3 - Low power modes overview PWR module API provides the capability to set low power mode constraints from various components or from the application. These constraints are provided to the SDK power manager. Upper layer (all Application code, connectivity stacks, etc.) can call directly the SDK Power Manger if it requires more advanced tuning. The PWR API can be found in PWR_Interface.h.

> **Note :** 'Upper layer' signifies all layers, applications, components, or modules that are above the connectivity framework in the Software architecture.
>
> **Note :** Each power domain has its own Low Power mode capability. The Low Power modes described below are for the main domain and it is supposed that the wake up domain goes to the same Low Power mode. This is not always true as the wake up domain that contains some wake up peripheral can go a lower Low Power mode state than the main domain so the peripherals in the wake up domain can remain operational when the main domain is in Low Power mode (deep sleep or power down modes). In this case, the context of the Hardware peripheral located in the wake up domain does not need to be saved and restored as for the peripherals located in the main domain

3.1 Wait for Interrupt (WFI) Definition

In the Wait for Interrupt (WFI) state, the CPU core is powered on, but is in an idle mode with the clock turned OFF.

Wake up time and typical use case

The wakeup time from this Low Power mode is insignificant because the Fast clock from FRO is still running.

This Low Power mode is mainly used when there is an hardware activity while the Software runs the Idle task. This allows the code execution to be temporarily suspende, thus reducing a bit the power consumption of the device by switching off the processor clock. When an interrupt fires, the processor clock is instantaneously restored to process the Interrupt Service Routine (ISR).

Usage

In order to prevent the software from programming the device to go to a lower Low Power mode (such as Deep Sleep, Power Down mode or Deep Power Down mode), the component responsible for the hardware drivers shall call PWR_SetLowPowerModeConstraint(PWR_WFI) function. When the Hardware activity is completed, the component shall release the constraint by calling PWR_ReleaseLowPowerModeConstraint(PWR_WFI).

Alternatively, the component can call PWR_LowPowerEnterCritical() and then PWR_LowPowerExitCritical() functions.

For fine tuning of the Low Power mode allowing more power saving, the component can call directly the SDK power manager API with PM_SetConstraints() function using the appropriate Low Power mode and low power constraint. However, this is reserved for more advanced user that knows the device very well. It is not recommended to do so.

The PWR module has no external dependencies, so the low-power entry and exit callback functions must be defined by the user for each peripheral that has specific low power constraints It is consequently convenient to register to the component the low power callbacks structure that is used for entering and exit low power critical sections. In Bluetooth LE, you can take the example in the app_conn.c file as shown here :

```
#if defined(gAppLowpowerEnabled_d) && (gAppLowpowerEnabled_d>0)
static const Seclib_LowpowerCriticalCBs_t app_LowpowerCriticalCBs =
{
  .SeclibEnterLowpowerCriticalFunc = &PWR_LowPowerEnterCritical,
  .SeclibExitLowpowerCriticalFunc  = &PWR_LowPowerExitCritical,
};
#endif

void BluetoothLEHost_Init(..)
{
  ...
      /* Cryptographic hardware initialization */
      SecLib_Init();
#if defined(gAppLowpowerEnabled_d) && (gAppLowpowerEnabled_d>0)
      /* Register PWR functions into SecLib module in order to disable device lowpower
         during Seclib processing. Typically, allow only WFI instruction when
         commands (key generation, encryption) are processed by Seclib  */
      SecLib_SetLowpowerCriticalCb(&app_LowpowerCriticalCBs);
#endif
  ...
}
```

Limitations

No limitation when using the WFI mode.

3.2 Sleep mode Sleep mode is similar to WFI low power mode but with some additional clock gating. The Sleep mode is device specific, please consult the Hardware reference manuel of the device for more information.

3.2 Deep Sleep mode **Definition**

In Deep Sleep mode, the fast clock is turned off, and the CPU along with the main power domain are placed into a retention state, with the voltage being scaled down to support state retention only. Because no high frequency clock is running, the voltage applied on the power domain can be reduced to reduce leakage on the hardware logic. This reduces the overall power consumption in the Deep Sleep mode. When waking up from Deep sleep mode, the core voltage is increased back to nominal voltage and the fast clock (FRO) is turned back on, the peripheral in this domain can be reused as normal.

To same more additional power, Some unused RAM banks can be powered off. this prevents from having current leakage and consequently, allow to reduce even more the power consumption in Deep SLeep mode. This is achieved by calling PLATFORM_SetRamBanksRetained() from low power entry function from board_lp.c file.

Usage

All firmware is able to implement Deep Sleep mode transparently to the application thanks to the PWR module, low power platform submodule and low power board file. This is described in the section Low-power implementation.

When entering this mode, it is recommended to turn the output pins into input mode, or high impedance to reduce leakage on the pads. This is typically done in pin_mux.c file, called from board.c file and executed from the low power callback in board_lp.c file. As an example, the TX line of the UART peripheral can be turned to disabled so it prevents the current from being drawn by the pad in Low Power mode.

Wake up time and typical use case

The wake up time is very fast, it takes mostly the time for the Fast FRO to start up again (couple of hundreds of microseconds) so this mode is a very good balance between power consumption in low-power mode and wake up latency and shall be used extensively in most of the use cases of the application.

Limitations

In Deep Sleep mode, the clock is disabled to the CPU and the main peripheral domain, so peripheral activity (for example, an on-going DMA transfer) is not possible in Deep Sleep mode.

3.3 Power Down mode

Definition

In Power Down mode, both the clock, and power are shut off to the CPU and the main peripheral domain. SRAM is retained, but register values are lost. The SDK power manager handles the restore of the processor registers and dependencies such as interrupt controller and similar ones transparently from the application.

Usage

The application, with the help of the low power board files, saves and restores the peripherals that were located in the power domain during the entry and exit of the power down mode. This is done from low power board_lp files in the entry/exit low power callbacks. Example is given for the serial manager and debug console in *board_lp.c* file in function *BOARD_ExitPowerDownCb()*.

If the device contains a dedicated wake up power domain where some wake up peripherals are located, if this wake up domain is not turned into power down mode but only Deep sleep mode or active mode, this peripheral does not need for a save and restore on low power entry/exit. For instance, on KW45, This is basically achieved when enabling the wakeup source of the peripheral *PWR_EnableWakeUpSource()* from *APP_ServiceInitLowpower()* function. Alternatively, this can be directly achieved by setting the constraint to the SDK power manager by calling *PM_SetConstraints()*, (use APP_LPUART0_WAKEUP_CONSTRAINTS for wakeup from UART constraint).

On exit from low power, The low power state of power domain can be retrieved by Platform API PLATFORM_GetLowpowerMode(). This API shall be called from low power exit callback function only.

As for Deep Sleep mode, software shall configure the output pins into input or high impedance during the Low Power mode to avoid leakage on the pads.

Wake up time and typical use case

The wake up time is significantly longer than wake up time from Deep Sleep (from several hundreds of micro-seconds to a couple of milliseconds depending on the platform). On some platform, it can takes longer, for instance, if ROM code is implemented and perform authentication checks for security and hardware logic in power domain needs to be restored (case for KW45).

However, After ROM code execution, the SDK power manager resumes the Idle task execution from where it left before entering low-power mode. Hence, the wakeup time from this mode is still significantly lower that the initialization time from a power on reset or any other reset.

Depending on the wakeup time of the platform and the low power time duration, This mode is recommended when no Software activity is expected to happen for the next several seconds. In Bluetooth LE, this mode is preferred in advertising or without Bluetooth LE activity. However, in scanning or connected mode, Regular wakes up happens regularly for instance to retrieve HCI message responses from the Link layer, the Deep Sleep mode is rather recommended.

Limitations

In addition to the Deep Sleep limitation (no Hardware processing on going when going to Power down mode) and the significant increase of the wake time, the Power Down mode requires the ROM code to execute and this last uses significant amount of memory in SRAM.

Typically, The first SRAM bank (16 KBytes) is used by the ROM code during execution so the Application firmware can use this section of SRAM for storing bss, rw data, or stacks. Only temporary data could be stored here and this location is overwritten on every Power Down exit sequence.

In order to avoid placing firmware data section (bss, rw, etc.) in the first SRAM bank, the linker script variable *gLowpowerPowerDownEnable_d* should be set to 1. Setting the linker script variable to avoid placing firmware data section in the first SRAM bank, The effect of setting this flag is to prevent the firmware from using the first 16 KB in SRAM.

> *Note* : This setting is ONLY required if the application implements Power Down mode.
> If Application uses other low-power mode, this is not required.

3.4 Deep Power-down mode

Definition

In Deep Power Down mode, the SRAM is not retained. This power mode is the lowest disponible, it is exited through reset sequence.

Usage

In addition to the Power Down limitation, the Deep Power Down mode shut down all memory in SRAM. Because it is exited through reset sequence the wake time is also longer.

Wake up time and typical use case

As this low-power mode is exited through the reset sequence, the wake up time is longer than any other mode. In Bluetooth LE, this mode is possible in no Bluetooth LE activity, and is preferred if we know that there will be no Bluetooth LE activity before a several amount of time.

Limitations

All memory in SRAM will be shut down in deep power down, the main limitation in going in this low-power mode is that the context will not be saved.

ModuleInfo

Overview The ModuleInfo is a small Connectivity Framework module that provides a mechanism that allows stack components to register information about themselves.

The information comprises :

- Component or module name (for example: Bootloader, IEEE 802.15.4 MAC, and Bluetooth LE Host) and associated version string
- Component or module ID
- Version number
- Build number

The information can be retrieved using shell commands or FSCI commands.

Detailed data types and APIs used in ConnFWK_APIs_documentation.pdf.

NVM: Non-volatile memory module

Overview In a standard Harvard-architecture-based MCU, the flash memory is used to store the program code and program constant data. Modern processors have a built-in flash memory controller that can be used under user program execution to store non-volatile data. The flash memories have individually erasable segments (sectors) and each segment has a limited number of erase cycles. If the same segments are used to store various kinds of data all the time, those segments quickly become unreliable. Therefore, a wear-leveling mechanism is necessary to prolong the service life of the memory. The NVM module in the connectivity framework provides a file system with a wear-leveling mechanism, described in the subsequent sections. The *NvIdle()* function handles the program and erase memory operations. Before resetting the MCU, *NvShutdown()* must be called to ensure that all save operations have been processed.

NVM boundaries and linker script requirement Most of the MCUs have only a standard flash memory that the non-volatile (NV) storage system uses. The amount of memory that the NV system uses for permanent storage and its boundaries are defined in the linker configuration file though the following linker symbols :

- NV_STORAGE_START_ADDRESS
- NV_STORAGE_END_ADDRESS
- NV_STORAGE_MAX_SECTORS
- NV_STORAGE_SECTOR_SIZE

The reserved memory consists of two virtual pages. The virtual pages are equally sized and each page is using one or more physical flash sectors. Therefore, the smallest configuration is using two physical sectors, one sector per virtual page.

NVM Table The Flash Management and Non-Volatile Storage Module holds a pointer to a RAM table. The upper layers of this table register information about data that the storage system should save and restore. An example of NVM table entry list is given below.

pData	ElemCount	ElemSize	EntryId	EntryType
0x1FFF9000	3	8	0xF1F4	MirroredInRam
0x1FFF7640	5	4	0xA2A6	NotMirroredInRam
0x1FFF1502	6	1	0x4212	NotMirroredInRam AutoRestore
0x1FFFF200	2	6	0x118F	MirroredInRam

NVM Table entry As show above, A NVM table entry contains a generic pointer to a contiguous RAM data structure, the number of elements the structure contains, the size of a single element, a table entry ID, and an entry type.

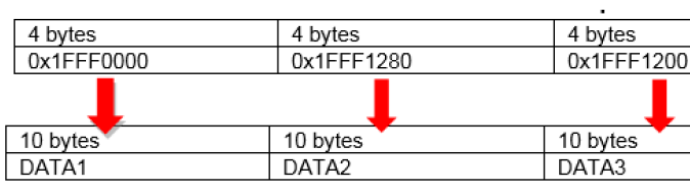
A RAM table entry has the following structure:

- pData (4 bytes) is a pointer to the RAM memory location where the dataset elements are stored.

- elemCnt (2 bytes) represents how many elements the dataset has.
- elemSz (2 bytes) is the size of a single element.
- entryID is a 16-bit unique ID of the dataset.
- dataEntryType is a 16-bit value representing the type of entry (mirrored/unmirrored/unmirrored auto restore).

For mirrored datasets, pData must point directly to the RAM data. For unmirrored datasets, it must be a double pointer to a vector of pointers. Each pointer in this table points to a RAM/FLASH area. Mirrored datasets require the data to be permanently kept in RAM, while unmirrored datasets have dataset entries either in flash or in RAM. If the unmirrored entries must be restored at the initialization, NotMirroredInRamAutoRestore should be used. The entryID gUnmirroredFeatureSet_d should be set to 1 for enabling unmirrored entries in the application. The last entry in the RAM table must have the entryID set to gNvEndOfTableId_c.

pData	0x1FFF8000
elemCnt	4
elemSz	10
entryID	1
dataEntryType	gNVM_NotM gNVM_NotM



The figure below provides an example of table entry :

When the data pointed to by the table entry pointer (pData) has changed (entirely or just a single element), the upper layers call the appropriate API function that requests the storage system to save the modified data. All the save operations (except for the synchronous save and atomic save) and the page erase and page copy operations are performed on system idle task. The application must create a task that calls NvIdle in an infinite loop. It should be created with OSA_PRIORITY_IDLE. However, the application may choose another priority. The save operations are done in one virtual page, which is the active page. After a save operation is performed on an unmirrored dataset, pData points to a flash location and the RAM pointer is freed. As a result, the effective data should always be allocated using the memory management module.

Active page The active page contains information about the records and the records. The storage system can save individual elements of a table entry or the entire table entry. Unmirrored datasets can only have individual saves. On mirrored datasets, the save/restore functions must receive the pointer to RAM data. For example, if the application must save the third element in the above vector, it should send 0x1FFF8000 + 2 * elemSz. For unmirrored datasets, the application must send the pointer that points to the area where the data is located. For example, if the application must save the third element in the above vector, it should send 0x1FFF8000 + 2 * sizeof(void*).

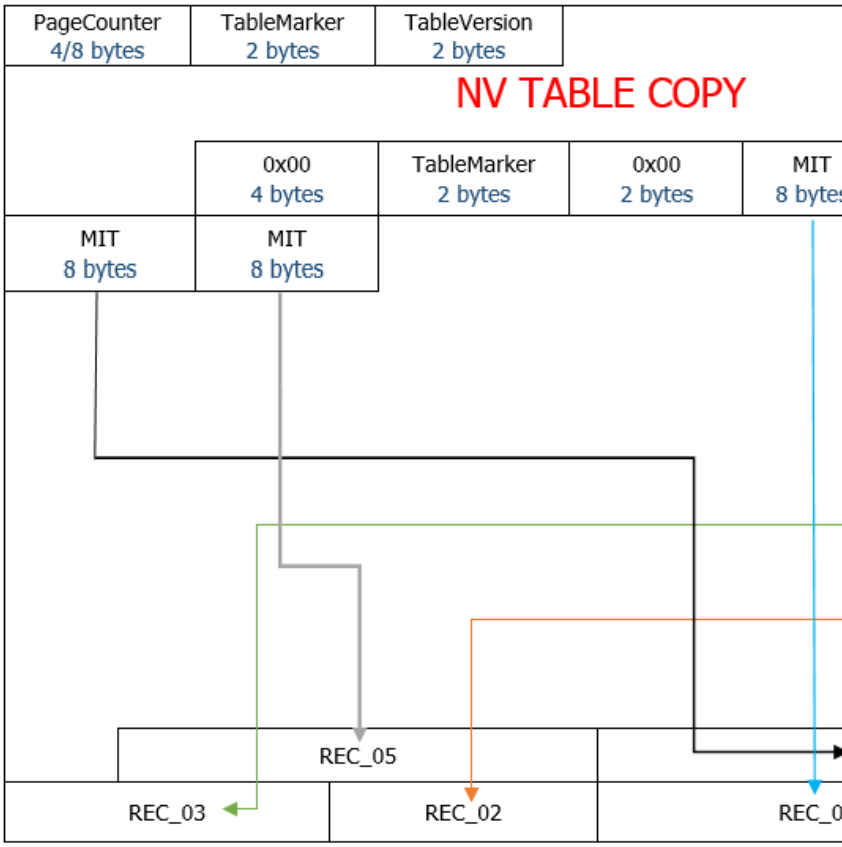
The page validity is guaranteed by the page counter. The page counter is a 32-bit value and is written at the beginning and at the end of the active page. The values need to be equal to consider the page a valid one. The value of the page counter is incremented after each page copy operation. A page erase operation is performed when the system is formatted. It is also performed when the page is full and a new record cannot be written into that page. Before being erased, the full page is first copied (only the most recent saves) and erased afterward.

The validity of the Meta Information Tag (MIT), and, therefore, of a record, is guaranteed by the MIT start and stop validation bytes. These two bytes must be equal to consider the record

referred by the MIT valid. Furthermore, the value of these bytes indicates the type of the record, whether it is a single element or an entire table entry. The nonvolatile storage system allows dynamic changes of the table within the RAM memory, as follows:

- Remove table entry
- Register table entry

A new table entry can be successfully registered if there is at least one entry previously removed or if the NV table contains uninitialized table entries, declared explicitly to register new table entries at run time. A new table entry can also replace an existing one if the register table entry is called with an overwrite set to true. This functionality is disabled by default and must be enabled by the application by setting gNvUseExtendedFeatureSet_d to 1.



The layout of an active page is shown below:

As shown above, the table stored in the RAM memory is copied into the flash active page, just after the table version. The "table start" and "table end" are marked by the table markers. The data pointers from RAM are not copied. A flash copy of a RAM table entry has the following structure:

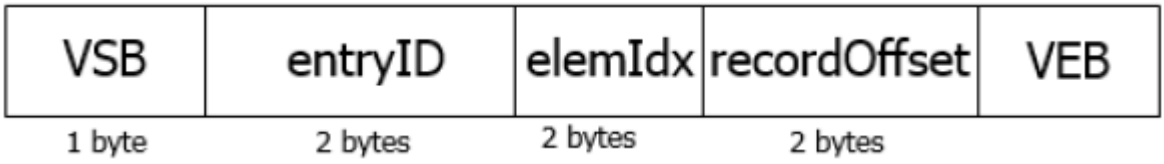
Where:

- entryID is the ID of the table entry
- entryType represents the type of the entry (mirrored/unmirrored/unmirrored auto restore)
- elemCnt is the elements count of that entry
- elemSz is the size of a single element

This copy of the RAM table in flash is used to determine whether the RAM table has changed. The table marker has a value of 0x4254 ("TB" if read as ASCII codes) and marks the beginning

and end of the NV table copy.

After the end of the RAM table copy, the Meta Information Tags (MITs) follow. Each MIT is used to store information related to one record. An MIT has the following structure:

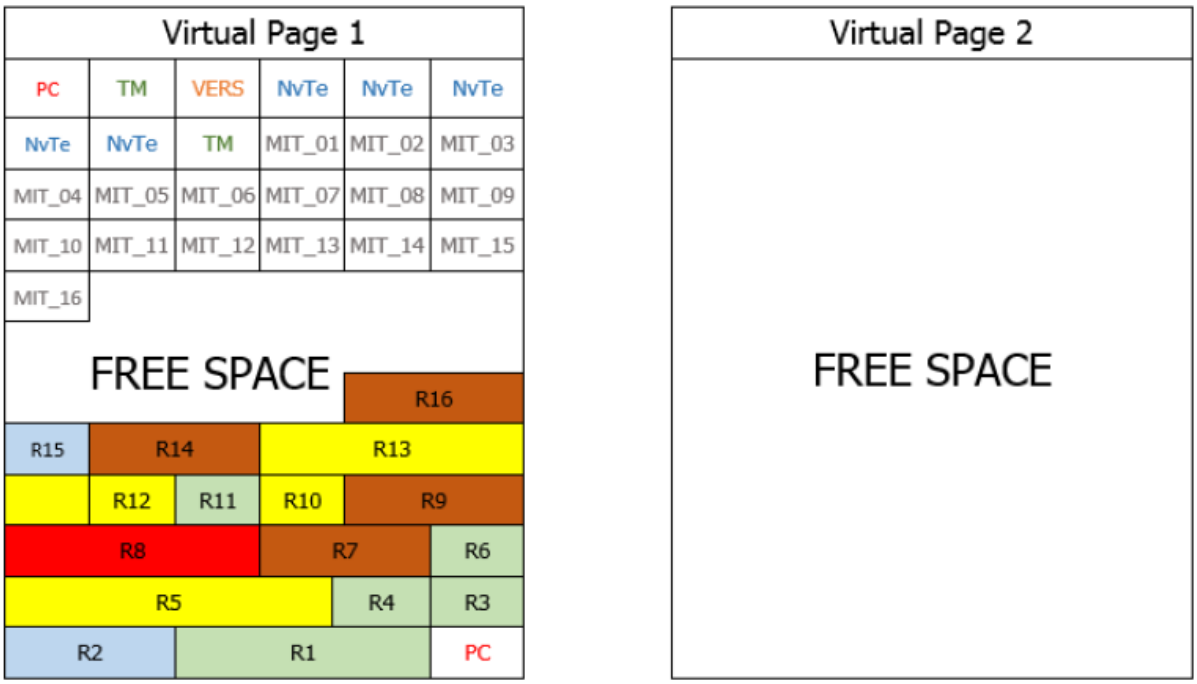


Where:

- VSB is the validation start byte.
- entryID is the ID of the NV table entry.
- elemIdx is the element index.
- recordOffset is the offset of the record related to the start address of the virtual page.
- VEB is the validation end byte.

A valid MIT has a VSB equal to a VEB. If the MIT refers to a single-element record type, VSB=VEB=0xAA. If the MIT refers to a full table entry record type (all elements from a table entry), VSB=VEB=0x55. Because the records are written to the flash page, the available page space decreases. As a result, the page becomes full and a new record does not have enough free space to be copied into that page.

In the example given below, the virtual page 1 is considered to be full if a new save request is pending and the page free space is not sufficient to copy the new record and the additional MIT. In this case, the latest saved datasets (table entries) are copied to virtual page 2.

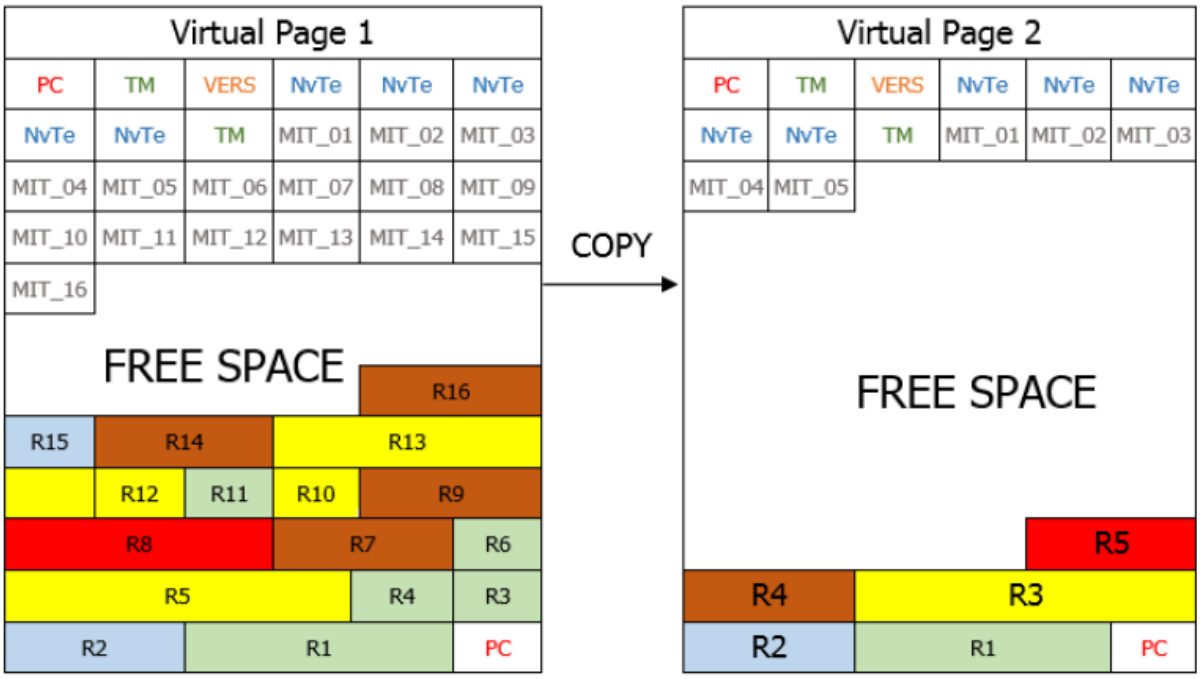


In this example, there are five datasets (one color for each dataset) with both 'full' and 'single' record types.

- R1 is a 'full' record type (contains all the NV table entry elements), whereas R3, R4, R6 and R11 are 'single' record types.
- R2 – full record type; R15 – single record type
- R5, R13 – full record type; R10, R12 – single record type

- R8 – full record type
- R7, R9, R14, R16 – full record type

As shown above, the R3, R4, R6, and R11 are 'single' record types, while R1 is a 'full' record type of the same dataset. When copied to virtual page 2, a defragmentation process takes place. As a result, the record copied to virtual page 2 has as much elements as R1, but individual elements are taken from R3, R4, R6, and R11. After the copy process completes, the virtual page 2 has five 'full' record types, one for each dataset. |This is illustrated below:



Finally, the virtual page 2 is validated by writing the PC value and a request to erase virtual page 1 is performed. The page is erased on an idle task, sector by sector where only one sector is erased at a time when idle task is executed.

If there is any difference between the RAM and flash tables, the application must call RecoverNvEntry for each entry that is different from its RAM copy to recover the entry data (ID, Type, ElemSz, ElemCnt) from flash before calling NvInit. The application must allocate the pData and change the RAM entry. It can choose to ignore the flash entry if the entry is not desired. If any entry from RAM differs from its flash equivalent at initialization, a page copy is triggered that ignores the entries that are different. In other words, data stored in those entries is lost.

The application can check if the RAM table was updated. In other words, if the MCU program was changed and the RAM table was updated, using the function GetFlashTableVersion and compare the result with the constant gNvFlashTableVersion_c. If the versions are different, NvInit detects the update and automatically upgrades the flash table. The upgrade process triggers a page copy that moves the flash data from the active page to the other one. It keeps the entries that were not modified intact and it moves the entries that had their elements count changed as follows:

- If the RAM element count is smaller than the flash element count, the upgrade only copies as many elements as are in RAM.
- If the RAM element count is larger than the flash element count, the upgrade copies all data from flash and fills the remaining space with data from RAM. If the entry size is changed, the entry is not copied. Any entryIds that are present in flash and not present in RAM are also not copied. This functionality is not supported if gNvUseExtendedFeatureSet_d is not set to 1.

ECC Fault detection The KW45/K32W1 internal flash is organized in 16 byte phrases and 8kB sectors (minimal erase unit). Its flash controller is synthesized so that it generates ECC information and an ECC generator / checker. During the programming of internal flash, errors may accidentally happen and cause ECC errors as a flash phrase is being written. These may happen due to multiple reasons:

- programmatic errors such as overwriting an already programmed phrase (transitioning bits from 0b to 1b). These are evitable by performing a blank check verification over phrase to be programmed, at the expense of processing power.
- occurrence of power drop or glitches during a programming operation.
- excessive wear of flash sector. The flash controller is capable of correcting one single ECC error but raises a bud fault whenever reading a phrase containing more than one ECC fault. Once an ECC error has 'infected' a flash phrase, the fault will remain and raise again at each read operation over the same phrase including blank check and prefetch. It can only be rid of by erasing the whole flash sector that contained the faulty phrase. In order to recover from situations where an ECC fault has occurred a gNvSalvageFromEccFault_d option has been added, which forces gNvVerifyReadBackAfterProgram_d to be defined to TRUE. If defined, the gNvVerifyReadBackAfterProgram_d option of the NVM module, causes the program to read back the programmed area after every flash programming operation. The verification is performed in safe mode if gNvSalvageFromEccFault_d is also defined. This is so as to detect ECC faults as early as possible as they appear, indeed when verifying a programming operation, one cannot be certain of the absence of ECC fault and avoid the bus fault. The safe API is thence used to perform the read back operation is performed using this safe API, so that we can tread in the flash and detect potential errors. The defects are detected on the fly whereas in the absence of safe read back, the error would cause a fault, potentially much later. During normal operation, assuming that no chip reset was provoked, this will consist in a single ECC fault either in the last record data or its meta information. Detecting such a fault calls for an immediate page copy to the other virtual page, so that the currently active page gets erased and the error gets cleared. Should the ECC fault occurs in the middle of a page copy operation, the switch of active page is postponed so that the fault page can be erased again and the copy can be restarted.

If the system underwent a power drop during a flash programming operation, sufficient to provoke a reset, at the ensuing reboot, ECC fault(s) may be present in the NVM area at the location that was being written. The detection is performed by an NVM sweeping mechanism, using the safe read API. That marks the faulty virtual page so that all subsequent reads within this virtual page are done with the safe API. If this case arises, a copy of the valid contents of the faulty page is attempted to the other virtual page. At NVM initialization, faults should be detected, either at the top of the meta data or at the bottom of the record area within the previous active page. This should guarantee that only the latest record write operation may be impaired. When the page copy has taken place, the faulty page is erased and the execution may resume. During NvCopyPage, when 'garbage collecting' occurs or whenever the current virtual active page needs to be transferred to the other virtual page, ECC errors are intercepted so that the operation can be attempted again in case of error. In case of NVM contents clobbering by programming errors, the salvage operation does its best to rescue as many records as possible but data will inevitably be lost.

An additional option -namely *gInterceptEccBusFaults_d* - was introduced in order to catch and correct ECC faults at Bus Fault handler level. Indeed, should an ECC bus fault fire, in spite of the precautions taken with NVM's gNvSalvageFromEccFault_d, we verify if the fault belongs to the NV storage. If so, a drastic policy can be adopted consisting in an erasure of the faulty sector. The corresponding Bus Fault handling is not part of the NVM, but dwells in the framework platform specific sources. Alternative handling could be implemented by the customer.

Save policy: Execution of program and erase operations on a flash an MCU core fetches code from cause perturbations of the core activity or requires to place critical code in RAM so that real-time ISR can still be served. The penalty of a sector erase is much higher than a simple program operation. The NVM is designed so as to limit the erase operations at 'garbage collecting' time,

so that flash wear is limited and no time is wasted. Several write policies are implemented to cope with the application constraints, one synchronous mode API and several posted write APIs. Among the posted write policies, the gNvmSaveOnIdleTimerPolicy_d compilation option selects a mode where flash write operations occur at time interval within the Idle task. Another option exists to 'randomize' the time interval with some jitter.

1) NvSyncSave performs a write synchronously with the disadvantage of stalling processor activity until comp
2) NvSaveOnCount posts a pending write operation and postpones the actual flash operation until number of record updates has reached a maximum. The actual write happens during Idle Task execution.see NvSetCountsBetweenSaves related API.
3) NvSaveOnInterval: posts a pending write operation and postpones the actual flash operation until the predefined number of ticks has elapsed. Optional mode - Active if (gNvmSaveOnIdleTimerPolicy_d & gNvmUseSaveOnTimerOn_c). see NvSetMinimumTicksBetweenSaves related API. Note that gNvmUseSaveIntervalJitter_c policy is a sub-option of gNvmSaveOnIdleTimerPolicy_d used to randomize slightly the time at which the write operation will happen.

Constant macro definition

- *gNvStorageIncluded_d* : If set to TRUE, it enables the whole functionality of the nonvolatile storage system. By default, it is set to FALSE (no code or data is generated for this module).
- *gNvUseFlexNVM_d* : If set to TRUE, it enables the FlexNVM functionality of the nonvolatile storage system. By default, it is set to FALSE. If FlexNVM is used, the standard nonvolatile storage system is disabled.
- *gNvFragmentation_Enabled_d* : Macro used to enable/disable the fragmented saves/restores (a particular element from a table entry can be saved or restored). It is set to FALSE by default.
- *gNvUseExtendedFeatureSet_d* : Macro used to enable/disable the extended feature set of the module:
  - Remove existing NV table entries
  - Register new NV table entries
  - Table upgrade

  It is set to FALSE by default.
- *gUnmirroredFeatureSet_d* : Macro used to enable unmirrored datasets. It is set to 0 by default.
- *gNvTableEntriesCountMax_c* : This constant defines the maximum count of the table entries (datasets) that the application is going to use. It is set to 32 by default.
- *gNvRecordsCopiedBufferSize_c* : This constant defines the size of the buffer used by the page copy function, when the copy operation performs defragmentation. The chosen value must be bigger than the maximum number of elements stored in any of the table entries. It is set by default to 64.
- *gNvCacheBufferSize_c* : This constant defines the size of the cache buffer used by the page copy function, when the copy operation does not perform defragmentation. The chosen value must be a multiple of 8. It is set by default to 64.
- *gNvMinimumTicksBetweenSaves_c* : This constant defines the minimum timer ticks between dataset saves (in seconds). It is set to 4 by default.
- *gNvCountsBetweenSaves_c* : This constant defines the number of calls to 'NvSaveOnCount' between dataset saves. It is set to 256 by default.

- *gNvInvalidDataEntry_c* : Macro used to mark a table entry as invalid in the NV table. The default value is 0xFFFFU.
- *gNvFormatRetryCount_c* : Macro used to define the maximum retries count value for the format operation. It is set to 3 by default.
- *gNvPendingSavesQueueSize_c* : Macro used to define the size of the pending saves queue. It is set to 32 by default.
- *gFifoOverwriteEnabled_c* : Macro used to enable overwriting older entries in the pending saves queue (if it is full). If it is FALSE and the queue is full, the module tries to process the oldest save in the queue to free a position. It is set to FALSE by default.
- *gNvMinimumFreeBytesCountStart_c* : Macro used to define the minimum free space at initialization. If the free space is smaller than this value, a page copy is triggered. It is set by default to 128.
- *gNvEndOfTableId_c* : Macro used to define the ID of the end-of-table entry. It is set to 0xFFFEU by default. No valid entry should use this ID.
- *gNvTableMarker_c* : Macro used to define the table marker value. The table marker is used to indicate the start and the end of the flash copy of the NV table. It is set to 0x4254U by default.
- *gNvFlashTableVersion_c* : Macro used to define the flash table version. It is used to determine if the NVM table was updated. It is set to 1 by default. The application should modify this every time the NVM table is updated and the data from NVM is still required.
- *gNvTableKeptInRam_d* : Set gNvTableKeptInRam_d to FALSE, if the NVM table is stored in FLASH memory (default). If the NVM table is stored in RAM memory, set the macro to TRUE.
- *gNvVerifyReadBackAfterProgram_d* : set by default force verification of NVM programming operations. Is forced implicitly when gNvSalvageFromEccFault_d is defined.
- *gNvSalvageFromEccFault_d* : use safe flash API to read from flash, and provide corrective action when ECC fault is met.

OtaSupport: Over-the-Air Programming Support

Overview This module includes APIs for the over-the-air image upgrade process. A Server device receives an image over the serial interface from a PC or other device thorough FSCI commands. If the Server has an image storage, the image is saved locally. If not, the image is requested chunk by chunk: With image storage

- OTA_RegisterToFsci()
- OTA_InitExternalMemory()
- OTA_WriteExternalMemory()
- ...
- OTA_WriteExternalMemory()

Without image storage:

- OTA_RegisterToFsci()
- OTA_QueryImageReq()
- OTA_ImageChunkReq()
- ...
- OTA_ImageChunkReq()

A Client device processes the received image by computing the CRC and filter unused data and stores the received image into a non-volatile storage. After the entire image has been transferred and verified, the Client device informs the Bootloader application that a new image is available, and that the MCU must be reset to start the upgrade process. See the following command sequence:

- OTA_StartImage()
- OTA_PushImageChunk() and OTA_CrcCompute ()
- ...
- OTA_PushImageChunk() and OTA_CrcCompute ()
- OTA_CommitImage()
- OTA_SetNewImageFlag()
- ResetMCU()

SecLib_RNG: Security library and random number generator

SecLib Implementation Flavors

Overview The NXP Connectivity Framework provides multiple SecLib implementations (flavors) to support different hardware capabilities, security requirements, and performance needs. Each flavor offers the same high-level cryptographic APIs but with different underlying implementations optimized for specific use cases.

Available SecLib Flavors

1. Software SecLib (SecLib.c)(RNG.c) **Description**: Pure software implementation providing cryptographic operations if no hardware is available.

Platforms: all platforms

Characteristics:

- Platform-independent software implementation
- No hardware acceleration requirements
- Consistent behavior across all platforms
- Higher CPU usage and power consumption
- Suitable for platforms without crypto hardware

Files: SecLib.c, SecLib_aes_mmo.c, RNG.c

2. EdgeLock SecLib (SecLib_sss.c)(RNG.c) **Description**: Hardware-accelerated implementation using NXP's EdgeLock secure subsystem s200.

Platforms: devices with S200: KW45, KW47, MCXW71, MCXW72

Characteristics:

- Hardware-backed security operations
- Secure key storage in hardware
- Hardware random number generation

- Lower CPU usage, better performance
- Enhanced security through hardware isolation
- Platform-specific (requires Secure Subsystem API)

Files: SecLib_sss.c, RNG.c

3. PSA Crypto SecLib (SecLib_psa.c)(RNG_psa.c) **Description**: Implementation based on ARM's Platform Security Architecture (PSA) Crypto API standard.

Platforms: MCXW23, (MCXW71 and MCXW72 for evaluation)

Characteristics:

- Industry-standard API
- Hardware acceleration when available
- Secure key management
- Future-proof and portable
- Compliance-ready

Files: SecLib_psa.c, SecLib_psa_config.h RNG_psa.c

4. MbedTLS SecLib (SecLib_mbedtls.c)(RNG_mbedTLS.c) **Description**: Implementation using the MbedTLS cryptographic library.

Platforms: all platforms, RW61

Characteristics:

- Well-tested and widely adopted
- Software and hardware acceleration support
- Extensive algorithm support
- Good performance optimization
- Large memory footprint

Files: SecLib_mbedTLS.c, SecLib_mbedtls_config.h, RNG_mbedTLS.c

Flavor Selection SecLib flavor is described in *'Kconfig'* and selected inside prj.conf or defconfig. Define one of this line to override the default configuration for your board:

```
CONFIG_MCUX_COMPONENT_middleware.wireless.framework.seclib_rng_port.sw=y
CONFIG_MCUX_COMPONENT_middleware.wireless.framework.seclib_rng_port.secure_subsystem=y
CONFIG_MCUX_COMPONENT_middleware.wireless.framework.seclib_rng_port.psa=y
CONFIG_MCUX_COMPONENT_middleware.wireless.framework.seclib_rng_port.mbedtls=y
```

The necessary configuration files are automatically added inside of CmakeLists.txt

Random number generator

Overview The RNG module is part of the framework used for random number generation. It uses hardware RNG peripherals as entropy sources (TRNG, Secure Subsystem, ...) to provide a true random number generator interface. A Pseudo-Random number generator (PRNG) implementation is available. The PRNG may depend of SecLib services (thus requiring a common mutex) to perform HMAC-SHA256, SHA256, AES-CTR, or alternateively a Lehmer Linear Congruential generator. A prerequisite for the PRNG to function with desired randomness is to be seeded using a proper source of entropy. If no hardware acceleration is present, the RNG may fallback to lesser quality ad-hoc source e.g if present SIM_UID registers, the UIDL is used as the initial seed for the random number generator.

Initialization The RNG module requires an initialization via a call to RNG_Init. The RNG initialization involves a call to RNG_SetSeed.

In the case of a dual core system consisting of a Host core and an NBU core, the Secure Subsystem is owned by the Host core. The Host core then has a direct access to its TRNG embedded in its secure subssystem. On the NBU code side, a request is emitted via RPMSG to the Host to provide a seed. On receipt of this request, the Host sets a 'reseed needed' flag (from the ISR context) If the core running the RNG service owns the TRNG entropy hardware (if any), it can get the seed directly form this hardware synchronously. In the case of an NBU that does not control the devices entropy source, that is owned by the Host, it request a seed from the Host processor via RPMSG exchange. On receipt of this request the Host sets a flag notifying of this request from the RPMSG ISR context.

RNG_ReInit API is to be used at wake up time in the context of LowPower. RNG_DeInit is used for unit tests and coverage purposes but has no useful role in a real application.

Asynchronous Seed Handling Enhancement The RNG module has been enhanced to use the framework's work queue system instead of polling from the IDLE thread.

Key Changes:

- **Work Queue Integration**: RNG_NotifyReseedNeeded() now submits seed operations to the work queue on Host cores
- **Non-blocking**: Replaces previous IDLE thread polling with asynchronous work queue scheduling
- **ISR-Safe**: Seed requests from ISR context are safely deferred to work queue execution

Benefits:

- Improved system responsiveness by removing IDLE thread dependency
- Better resource management through work queue scheduling

Seed handling RNG_SetSeed: RNG_SetExternalSeed may be used to inject application entropy to RNG context seed using a supplied array of bytes. RNG_IsReseedNeeded used from task in Host core to check whether seed must be sent to NBU core.

RNG_GetTrueRandomNumber is the API used to generate a Random 32 bit number from a HW source of entropy. It is essential if only to seed the pseudo random number generator.

RNG_GetPseudoRandomData is used to generate arrays of random bytes.

RNG_NotifyReseedNeeded Enhanced to use work queue submission on Host cores instead of requiring IDLE thread polling. Sets reseed flag and schedules asynchronous seed generation.

Security Library

Overview The framework provides support for cryptography in the security module. It supports both software and hardware encryption. Depending on the device, the hardware encryption uses either the S200, MMCAU, LTC, or CAU3 module instruction set or dedicated AES and SHA hardware blocks.

Software implementation is provided in a library format.

Support for security algorithms

	SW Seclib : SecLib.c	EdgeLock SecLib_sss.c	Seclib_p	Mbedtls SecLib_mbec	nccl (part of SecLib.c)	Usage example
AES_128	SecLib_aes.c	x	x	x		
AES_128_ECB		x	x	x		
AES_128_CBC	x	x		x		
AES_128_CTR encryption	x	x				
AES_128_OFB Encryption	x					
AES_128_CMAC	x	x	x	x		BLE connection, ieee 15.4
AES_128_EAX	x					
AES_128_CCM	x	x	x	x		BLE, ieee 15.4
SHA1	SecLib_sha.c	x		x		
SHA256	x	x	x	x		
HMAC_SHA256	x	x		x		PRNG, Digest for Matter
ECDH_P256 shared secret generation	x (by 15 incremental steps) -> SecLib_ecdh.c	x with MACRO SecLibECDHUseSSS	x	x	x	BLE pairing,
EC_P256 key pair generation	x	x	x	x	x	
EC_P256 public key generation from private key				x	x	Matter (ECDSA)
ECDSA_P256 hash and msg signature generation / verification		only if owner of the key pair		x	x	Matter
SPAKE2+ P256 arithmetics				x	x	Matter

SecLib_psa The framework provides PSA (Platform Security Architecture) Crypto API support through the `SecLib_psa.c` implementation. This provides a standardized cryptographic interface

that can leverage hardware acceleration when available.

Configuration PSA support is configured through the SecLib_psa_config.h file, which defines:

- Hardware acceleration capabilities via MBEDTLS_PSA_ACCEL_* macros
- Memory management integration with the framework's memory manager
- External RNG integration

Advantages of PSA Crypto API

Standardization and Portability

- **Vendor Independence**: Applications using PSA APIs can be easily ported between different hardware platforms without changing cryptographic code
- **Industry Standard**: PSA Crypto API is an ARM-defined industry standard (PSA Certified), ensuring consistent behavior across different platforms and vendors
- **Future-Proof**: As an evolving standard, PSA provides long-term compatibility and support for emerging cryptographic requirements

Security Benefits

- **Secure Key Management**: PSA provides built-in secure key storage and lifecycle management, reducing the risk of key exposure
- **Hardware Security Integration**: Seamless integration with hardware security modules (HSMs) and secure elements when available
- **Isolation**: Keys and cryptographic operations can be isolated from application code, reducing attack surface
- **Tamper Resistance**: Hardware-backed implementations provide protection against physical attacks

BLE advanced secure mode

New elements in existing structures: computeDhKeyParam_t::keepInternalBlob - boolean telling if the shared blob is kept in this structure(in .outpoint) after ECDH_P256_ComputeDhKey() or ECDH_P256_ComputeDhKeySeg() call.

New arguments in existing functions: ECDH_P256_ComputeDhKey keepBlobDhKey - boolean telling ECDH_P256_ComputeDhKey() or ECDH_P256_ComputeDhKeySeg() to keep the shared object after computation for later use (it is required by the SecLib_GenerateBluetoothF5KeysSecure).

New macros: gSecLibSssUseEncryptedKeys_d - Enable or disable S200 blobs SecLib support. 0 - the Bluetooth Keys are available in plaintext, 1 - the Bluetooth Keys are not available in plaintext, but in secured blobs. Default is disabled.

New functions:

LE Secure connections pairing:

void ECDH_P256_FreeDhKeyDataSecure This is a function used to free the shared object stored in computeDhKeyParam_t. When user calls ECDH_P256_ComputeDhKeySeg() with keepBlobDhKey set to 1, it should also call **ECDH_P256_FreeDhKeyDataSecure** .

SecLib_GenerateBluetoothF5Keys This function is extracted from the Bluetooth LE Host Stack implementation. This corresponds to the legacy implementation without key blobs.

SecLib_GenerateBluetoothF5KeysSecure Similar to **SecLib_GenerateBluetoothF5Keys** this function is modified to work with key blobs, the reason is to not use SSS inside the Bluetooth LE Host Stack.

SecLib_DeriveBluetoothSKD This is a helper function used by the Bluetooth LE Host Stack in the pairing procedure, when receiving the vendor HCI command specifying that the ESK needs to be provided to LL.

ELKE_BLE_SM_F5_DeriveKeys This is a private function, helper for **SecLib_GenerateBluetoothF5KeysSecure**. It was provided by the STEC team.

Privacy:

SecLib_ObfuscateKeySecure This is a function used by the Bluetooth LE Host Stack to obfuscate the IRK before setting it to Bluetooth LE Controller or before saving it to NVM

SecLib_DeobfuscateKeySecure This is a function used by the Bluetooth LE Host Stack to extract the plaintext IRK key from the saved NVM blob.

SecLib_VerifyBluetoothAh This function is extracted from the legacy Bluetooth LE Host Stack implementation using plaintext keys.

SecLib_VerifyBluetoothAhSecure Similar to **SecLib_VerifyBluetoothAh** with modification to work with S200 key blob.

SecLib_GenerateSymmetricKey This is a function used by the application to generate the local IRK and local CSRK.

SecLib_GenerateBluetoothEIRKBlobSecure This is a function used by the application to generate the EIRK needed by Bluetooth LE Controller from the IRK blob.

A2B feature

ECDH_P256_ComputeA2BKey This function is used to compute the EdgeLock to EdgeLock key. pInPeerPublicKey points to the peer public key, pOutE2EKey is the pointer to where the E2E key object will be stored, this will be freed by the application when it is no longer required by calling ECDH_P256_FreeE2EKeyData().

ECDH_P256_FreeE2EKeyData This function is used to free the key object given as a parameter. It is used by the application to free the E2E key when is no longer needed.

SecLib_ExportA2BBlobSecure This function is used to import an ELKE blob or plain text symmetric key in s200 and export an E2E key blob. The input type is identified by the keyType parameter.

SecLib_ImportA2BBlobSecure This function is used to import an E2E key blob in s200 and export an ELKE blob or plain text symmetric key. The output type is identified by the keyType parameter.

LE Secure connections Pairing flow and SecLib usage:

1. Each device needs to generate locally the public+private keypair. This is done using **ECDH_P256_GenerateKeys**.
2. Devices exchange their public keys.
3. Upon receiving the peer device's public key, local device is computing DH key using **ECDH_P256_ComputeDhKey**.
4. Each device sends DHKeyCheck packet
5. Upon receiving DhKeyCheck each device computes LTK blob using **SecLib_GenerateBluetoothF5Keys**
6. After computing the each device sends HCI_LeStartEnc (on initiator), HCI_Le_Provide_Long_Term_Key (on responder)
7. Bluetooth LE Controller sends back SKD report custom event
8. Bluetooth LE Host Stack computes ESKD based on LTK blob using **SecLib_DeriveBluetoothSKD** and sends it to Bluetooth LE Controller
9. Bluetooth LE Controller encrypts the link

IRK flow and SecLib usage:

1. At startup, when gInitializationComplete_c event is received:

- the local IRK is generated using **SecLib_GenerateSymmetricKey**
- the local EIRK is generated using **SecLib_GenerateBluetoothEIRKBlobSecure**
- local CSRK is generated using **SecLib_GenerateSymmetricKey**

2. During legacy pairing when receiving bonding keys, IRK is obfuscated using **SecLib_ObfuscateKeySecure** and stored
3. When app wants to set the OOB keys using Gap_SaveKeys the IRK is obfuscated using **SecLib_ObfuscateKeySecure**
4. When application calls API Gap_VerifyPrivateResolvableAddress IRK is obfuscated using **SecLib_ObfuscateKeySecure** and verified using **SecLib_VerifyBluetoothAhSecure**
5. When a new connection is received in Host with RPA address not resolved by the Bluetooth LE Controller, the Host tries to resolve it by obfuscating it using **SecLib_ObfuscateKeySecure** and verifying it using **SecLib_VerifyBluetoothAhSecure**
6. When adding a peer in Bluetooth LE Controller resolving list, the peer's IRK is obfuscated using **SecLib_ObfuscateKeySecure** before setting it using **HCI_Le_Add_Device_To_Resolving_List**.

7. When an IRK plaintext is requested by the application using Gap_LoadKeys it is obtained using **SecLib_DeobfuscateKeySecure**
8. When legacy pairing completes and LTK needs to be send in the pairing complete event (gConnEvtPairingComplete_c) the **SecLib_DeobfuscateKey** is used to extract the plaintext.

A2B flow and SecLib usage:

1. At startup, when gInitializationComplete_c event is received, the application will call **ECDH_P256_GenerateKeys** to generate the public/private key pair required for the E2E key derivation and send the public key to the peer device.
2. When the public key is received from the peer device, the application will call **ECDH_P256_ComputeA2BKeySecure** to generate the EdgeLock to EdgeLock key.
3. The application will obtain an E2E IRK blob by calling **SecLib_ExportA2BBlobSecure** with key type gSecElkeBlob_c. The obtained blob is sent to the peer anchor. The peer anchor will call **SecLib_ImportA2BBlob** with keyType gSecElkeBlob_c and save the resulting ELKE blob in NVM, for Digital Key both anchors must have the same IRK.
4. After pairing, in order to send the LTK and IRK contained in the bonding data securely, the application will call **SecLib_ExportA2BBlobSecure** with keyType gSecLtkElkeBlob_c for the LTK, and **SecLib_ExportA2BBlobSecure** with keyType gSecPlainText_c for the IRK. The E2E blobs obtained are sent along with the rest of the bonding data to the peer anchor device.
5. After the bonding data is trasfered the E2E key is no longer needed and **ECDH_P256_FreeE2EKeyData** is called with the key object obtained at step 2 when **ECDH_P256_ComputeA2BKeySecure** was called.

Sensors

Overview The Sensors module provides an API to communicate with the ADC. Two values can be obtained by this module :

- Temperature value
- Battery level

The temperature is given in tenths of degrees Celsius and the battery in percentage.

This module is multi-caller, the ADC is protected by a mutex on the resource and by prevententing lowpower (only WFI) during its processing. Platform specific code can be find in fwk_platform_sensors.c/h.

Features

Manual Measurement Calling “SENSORS_GetTemperature()” or “SENSORS_GetBatteryLevel()” function will give the last measured value that was copied to RAM. If you want to refresh this value you must first call the trigger function (SENSORS_TriggerTemperatureMeasurement() / SENSORS_TriggerBatteryMeasurement()) that configures the ADC to perform temperature or battery measurement, then a call to the refresh function (SENSORS_RefreshTemperatureValue() / SENSORS_RefreshBatteryLevel()) will read the returned value from the ADC and save it in RAM. You can run code between these two functions because the ADC won’t give you the value until it’s fully ready.

> **Important**: When a measurement is ongoing, new measurement requests will be **dropped**. Measurement triggers are only accepted when the measurement state is **IDLE**. This applies to both temperature and battery measurements to prevent ADC resource conflicts.

Periodic Temperature Measurement The Sensors module supports automatic periodic temperature measurements that can be requested from two sources:

1. HOST Requests Applications can request periodic temperature measurements using:

```
SENSORS_TriggerPeriodicTemperatureMeasurement(uint32_t temperature_meas_interval_ms);
```

2. NBU Requests (wireless_mcu platforms only) The NBU (Narrow Band Unit) can request periodic temperature measurements. For more details about NBU temperature requests check NBU Integration Features section bellow.

Periodic Measurement Behavior **Immediate Trigger**: Every periodic request (HOST or NBU) triggers an **immediate measurement** before starting the interval timer.

Interval Management:

- **Minimum Interval Selection**: When both HOST and NBU request periodic measurements, the system uses the **shorter interval**
- **Dynamic Updates**: Changing the interval from either source immediately triggers a new measurement and restarts the timer with the new effective interval
- **Independent Control**: HOST and NBU can independently start/stop their periodic requests

One-shot Mode: Setting interval to 0 triggers a single measurement and stops periodic operation for that source.

Usage Examples

```
// HOST requests periodic measurement every 100ms
SENSORS_TriggerPeriodicTemperatureMeasurement(100);

// HOST requests one-shot measurement and stops periodic operation
SENSORS_TriggerPeriodicTemperatureMeasurement(0);
```

Concurrent Requests Behavior

HOST Interval	NBU Interval	Effective Interval	Notes
200ms	100ms	100ms	NBU interval is shorter
100ms	300ms	100ms	HOST interval is shorter
0 (stopped)	150ms	150ms	Only NBU active
200ms	0 (stopped)	200ms	Only HOST active
0 (stopped)	0 (stopped)	No periodic	Both stopped

Measurement State Management The Sensors module uses a state machine to manage ADC resource access:

States:

- **MEASUREMENT_IDLE**: No measurement in progress, ready to accept new requests
- **TMP_MEASUREMENT_ONGOING**: Temperature measurement in progress
- **BAT_MEASUREMENT_ONGOING**: Battery measurement in progress

Behavior:

- **Request Acceptance**: New measurement requests are only processed when state is MEASUREMENT_IDLE
- **Request Dropping**: If a measurement request is made while another measurement is ongoing, the new request is silently dropped
- **State Transitions**: State changes from IDLE ONGOING when measurement starts, and back to IDLE when measurement completes
- **Recovery Mechanism**: For periodic measurements, if a trigger fails due to ongoing measurement, a 1ms retry timer is started

NBU Integration Features

Platform Support: NBU integration features are only supported on **wireless_mcu** platforms. These features are **not available** on RT, mcxw23, and rw61x platforms.

Periodic Temperature Measurement from NBU The Sensors module supports automatic periodic temperature measurements triggered by NBU requests. This feature allows the system to automatically measure temperature at regular intervals without manual intervention. This temperature data is used for XTAL32M calibration and Channel Sounding calibration.

How it works:

1. **NBU Request**: The NBU can request periodic temperature measurements by calling the registered callback with a measurement interval
2. **Timer-based Scheduling**: A timer manager is used to schedule periodic measurements based on the requested interval
3. **Workqueue Processing**: Temperature measurement triggers are processed through the system workqueue to avoid blocking operations
4. **Automatic Refresh**: Once a measurement is complete, the next measurement is automatically scheduled

Key Components:

- **Temperature Request Callback**: Sensors_TemperatureReqCb() - Handles NBU requests for periodic measurements
- **Timer Callback**: Sensors_TempMeasTimerCallback() - Triggers measurements on timer expiration
- **Work Handler**: Sensors_PeriodicTempWorkHandler() - Processes temperature measurement requests in workqueue context
- **Ready Callback**: Sensors_TemperatureReadyCb() - Called when temperature measurement is complete

Usage:

- **One-shot measurement**: NBU requests with interval = 0
- **Periodic measurement**: NBU requests with interval > 0 (in milliseconds)
- The periodic measurement continues until a new request changes the interval or stops it

Temperature Filtering and NBU Communication

Modified Moving Average Filter On wireless_mcu platforms, temperature readings are processed through a Modified Moving Average (MMA) filter to reduce noise and prevent unnecessary NBU notifications. This filter helps to:

- **Limit XTAL32M calibration changes**: Prevents excessive calibration adjustments that could degrade RF performances
- **Optimize power consumption**: Reduces NBU wake-up events by limiting notifications to significant temperature changes
- **Reduce noise**: Smooths out temperature reading fluctuations for more stable system behavior

The XTAL32M calibration process uses a LUT (Look-Up Table) that contains pre-characterized calibration values mapping temperature ranges to optimal XTAL32M frequency adjustments, enabling precise crystal oscillator tuning based on thermal conditions.

The LUT is registered using `PLATFORM_RegisterXtal32MTempCompLut()` and this is currently done in `examples/_common/project_segments/wireless/wireless_mcu/app_common/hardware_init.c` and requires `gBoardUseXtal32MTempComp` to be enabled.

Configuration: Filter size is controlled by `PLATFORM_TEMPERATURE_FILTER_SIZE` (default: 4 samples)

NBU Temperature Threshold Temperature values are only sent to the NBU when they exceed a configurable threshold:

Threshold is Controlled by `PLATFORM_TEMP_SENT_NBU_THRESHOLD` (default: 10 tenths of degrees Celsius = 1°C)

Temperature Processing Flow:

1. **Raw Measurement**: ADC interrupt captures temperature reading
2. **Sensors Module Informed** Sensors referesh the temperature value
3. **Filtering**: Raw value is processed through MMA filter
4. **Threshold Check**: Filtered value is compared against last sent value ± threshold
5. **NBU Communication**: If threshold is exceeded:
    - XTAL32M calibration is performed (if LUT is available)
    - Temperature value is sent to NBU
    - Last sent value is updated

```
Temperature Request
        ↓
ADC Interrupt - Measurement is ready
        ↓
Raw Temperature Reading
```

(continues on next page)

(continued from previous page)

```
          ↓
Sensors Module Informed - Temperature refresh
          ↓
MMA Filter Processing
          ↓
Threshold Check: |filtered - last_sent| > threshold?
     ↓                    ↓
    YES                   NO
     ↓
LUT Available?
  ↓       ↓
 YES      NO
  ↓       ↓
XTAL32M     ↓
Calibration ↓
  ↓       ↓

     ↓
Send Temperature to NBU
     ↓
Update Last Value Sent to NBU
```

Platform Configuration Macros:

```
// Temperature filter size (number of samples for averaging)
#define PLATFORM_TEMPERATURE_FILTER_SIZE 4U

// Threshold for sending temperature to NBU (in tenths of degrees Celsius)
#define PLATFORM_TEMP_SENT_NBU_THRESHOLD 10  // 1°C
```

API Usage Examples

Manual Temperature Measurement

```
int32_t temperature1, temperature2;
SENSORS_TriggerTemperatureMeasurement();
...
temperature1 = (SENSORS_RefreshTemperatureValue())/10;
temperature2 = (SENSORS_GetTemperature())/10;
```

In this case temperature1 is equal to temperature2, both are in degree celsius.

Battery Level Measurement

```
uint8_t batteryLevel;
SENSORS_TriggerBatteryMeasurement();
...
batteryLevel = SENSORS_RefreshBatteryLevel();
// batteryLevel is now in percentage (0-100%)
```

Handling Concurrent Requests

```
// First request - will be accepted
SENSORS_TriggerTemperatureMeasurement();

// Second request while first is ongoing - will be dropped silently
```

(continues on next page)

(continued from previous page)

```
SENSORS_TriggerBatteryMeasurement();  // This will be ignored

// Wait for first measurement to complete
int32_t temp = SENSORS_RefreshTemperatureValue();

// Now battery measurement can be triggered
SENSORS_TriggerBatteryMeasurement();  // This will be accepted
```

Low Power Optimization - Wake-up Temperature Measurement

```
// In low power exit callback (interrupts masked, scheduler disabled)
/* Trig temperature measurement on wake-up.
 * If a temperature change is detected,
 * the new value will be sent to the NBU for appropriate XTAL32M trimming.
 * During channel sounding measurements, this temperature is also used for RTT compensation.
 * Since this function runs with interrupts masked and the scheduler disabled,
 * the unsafe variant must be used to avoid mutex acquisition.
 */
SENSORS_TriggerTemperatureMeasurementUnsafe();
```

This approach optimizes power consumption and provides additional benefits:

- **XTAL32M Trimming**: Temperature changes are sent to NBU for crystal oscillator calibration only if they exceed the configured threshold after filtering
- **RTT Compensation**: This temperature data is used by the NBU for RTT compensation during channel sounding measurements
- **Safe Execution**: Uses the unsafe variant to avoid mutex acquisition in interrupt context with scheduler disabled In case of periodic measurement:
- **Power Optimization**: Triggers temperature measurements when the system is already awake, avoiding dedicated timer wake-ups
- **Timer Reset**: Resets the periodic measurement timer to prevent unnecessary wake-ups

  Note: This optimization is implemented by default in wireless platform examples when gAppUseSensors_d is enabled. The temperature measurement is automatically triggered in the low power exit callback (BOARD_ExitLowPowerCb()) in examples/_common/project_segments/wireless/wireless_mcu/board_lp.c.

Constant macro definitions

Name :

```
#define VALUE_NOT_AVAILABLE_8 0xFFu
#define VALUE_NOT_AVAILABLE_32 0xFFFFFFFFu
```

Description :

Defines the error value that can be compared to the value obtain on the ADC.

SFC : Smart Frequency Calibration

Overview The Smart Frequency Calibration module provides operations and calibration for the FRO32K source clock. This module is split between main core and Radio core:

- fwk_rf_sfc.[ch]: RF_SFC module on Radio core that provides Main FRO32K measurement/calibration and state machine in synchornization with Radio domain activities. See details below.

- fwk_sfc.h: SFC module on host core that provides type definition for usage with fwk_platform_ics.[ch] with PLATFORM_FwkSrvSetRfSfcConfig() API and fwk_platform_ble.c for received callback from the NBU core

Host SFC Module

Algorithm parametrization This module provides ability to configure the RF_SFC module by sending message to Radio core through fwk_platform_ics.c PLATFORM_FwkSrvSetRfSfcConfig():

- Filter size
- Maximum ppm threshold
- Maximum calibration interval
- Number of sample in filter to swicth from convergence to monitor mode

Ppm target The ppm target is the deviation from the target clock accepted by the algorithm. When the deviation is larger than the ppm target. The algorithm will update the trimming value and reset the filter. The ppm target cannot be more aggressive RF_SFC_MAXIMAL_PPM_TARGET in order to avoid having to update trimming value at each measurement.

Filter size Filter size must be included between RF_SFC_MINIMAL_FILTER_SIZE and RF_SFC_MAXIMAL_FILTER_SIZE. See *Filtering and Frequency estimation* section for more details on the parameter.

Maximum calibration interval In monitor mode, new measurement are triggered by low-power entry/exit. If the NBU core has a lot of radio activity it could never enter lowpower. The maximum calibration interval is here to ensure a measurement is done regularly. When executing idle the SFC module checks when the last measurement has been done, if it has been too long, it reset the filter and forces a new measurement

Trig sample number The trig sample number is the number of samples needed by the algorithm in its filter to switch from convergence to monitor mode. Having more than one sample in convergence mode allows to confirm the trimming value that we have set.

SFC debug information On the other way, the RF_SFC from Radio core sends back notifications to SFC module on main core using RX callback PLATFORM_RegisterFroNotificationCallback() from fwk_platform_ics.h and such information:

- last measured frequency
- average ppm from 32768Khz frequency
- last ppm measured from 32768Khz frequency
- FRO trimming value

RF_SFC module The RF_SFC module provides the functionality to calibrate the FRO32K source clock during Initialization and radio activity.

The RF_SFC is mostly used on XTAL32K less solution when no 32Khz crystal is soldered on the board. It allows to calibrate the FRO32K source clock to the desired frequency to keep Radio time base within the allowed tolerance given by the connectivity standards. However, even on a XTAL32K solution, the RF_SFC is also used during Initialization until the XTAL32K is up and

running in the system. The system firstly runs on the FRO32K clock source then switch to the XTAL32K clock source when it is ready with enough accuracy. This allows to save significant boot time as the FRO32K start up (including calibration) is much faster compared to XTAL32K .

This module will handle:

- FRO32K clock frequency measurement against 32Mhz crystal. It schedules appropriately the start of the measurement and gets the result when completed,
- Filter and estimate the 32Khz frequency value and error by averaging from the last measurements,
- FRO32K calibration in order to update the trimming value to reduce the frequency error on the clock.

The targeted frequency offset shall be within 200ppm. The RF_SFC will handle two modes of operation:

- Convergence mode: when frequency estimation is above 200pm,
- Monitor mode: when frequency estimation is below 200pm.

The RF_SFC module works in active and all low power modes on NBU domain, or on host application domain except power down mode. Power down mode on host application domain is not supported with the FRO32K configuration as clock source.

Feature enablement Enabling the FRO32K is done by calling the PLATFORM_InitFro32K() function during application initialization in hardware_init.c file, in BOARD_InitHardware() function. If FRO32K is not enabled, Oscillator XTAL32K shall be called instead by calling PLATFORM_InitOsc32K() function. The call to PLATFORM_InitFro32K() from BOARD_InitHardware() can be done by setting the Compilation flag gBoardUseFro32k_d to 1 in hardware_init.c or any header files included from this file.

```
#define gBoardUseFro32k_d        1
```

Detailed description

Frequency measurements When NBU low power is enabled, the frequency measurements are triggered on Low power wake-up by HW signal. The SFC process called from Idle task will check regularly the completion of the frequency measurement. When the measurement is done, it goes to filtering and frequency estimation process. The frequency measurement duration depends on monitor mode or convergence mode: In convergence mode, the frequency measurement duration is 0.5ms while it is 2ms in monitor mode. In monitor mode, the duration value remains less than the minimal radio activity duration so it does not impact the low power consumption in monitoring mode.

Filtering and Frequency estimation The FRO32KHz frequency measurement values are noisy because of thermal noise on the FRO32K itself. Also, the frequency measurement can introduce some error. In monitoring mode, it is required to filter the measurements by applying an exponential filter. new_estimation = (new_measurement + ((1 « n) -1) * last_estimation) » n

Default value for n is 7 (meaning 128 samples in the averaging window).

Frequency calibration When the frequency estimation gets higher than the targeted 200ppm target, the RF_SFC updates the trimming value for one positive or negative increment. For this purpose, it requires to:

- wake up the host application domain and keep the domain active,

- update the trim register of the FRO32K , this register is used to trim the capacitance value of the FRO32K,
- re-allow the host application domain to enter low power.

A slight power impact is expected during a calibration update due to host domain wake-up.

Operational modes When the low power mode is enabled on NBU power domain, RF SFC handles two modes of operation: convergence and monitor modes. However, when low power is disabled on NBU power domain, only convergence mode is supported.

Convergence mode Convergence mode is used when the estimated FRO32K frequency is above 200ppm or when the filter has been reset. Typically this occurs :

- During Power ON reset or other reset when NBU is switched OFF
- When temperature varies and FRO32K frequency deviates outside 200ppm threshold target
- When no calibration has been done during some time as we discard old values that could influence the algorithm

The convergence mode process typically starts with a FRO32K trim register update, performs a frequency measurement and the FRO32K trim register is updated until the measured frequency gets below 200ppm. These operations are repeated in a loop until the estimated frequency value gets below 200ppm. When below 200ppm during multiple measurements, the RC SFC switches to Monitoring mode. The convergence mode is only a transition mode to monitoring mode. In convergence mode, the NBU power domain does not go to low power. The convergence mode time duration depends on the initial frequency error of the FR032K. Default frequency measurement duration is 0.5ms so 20 measurements (given as example only) will require less than 10 ms to converge.

Monitoring mode Monitoring mode is used when the estimated FRO32K frequency is below 200ppm. In this mode, the measurement is triggered on NBU domain wake up from low power mode using an internal hardware signal. The exponential filter is applied to compute the frequency estimation. If the frequency estimation value is still within 200ppm, the NBU power domain is allowed to go to low power. If the estimated value gets above the 200ppm threshold, the RF SFC switch back to convergence mode. The trim register is updated by one increment (positive or negative) and because the frequency has been adjusted and changed, the estimated filtered frequency is reset to discard all previous measurements. Going back to convergence mode typically happens during a temperature gradient. If the temperature is constant, it is not expected to have the estimated value to go beyond 200ppm so no calibration should be required.

Initialization and configuration During initialization, the RF SFC module will block the Radio Software until monitoring mode is reached. This is to prevent the radio from running with an inaccurate time base due to an important 32k clock frequency error.

Initialization and configuration is done by the NBU core. The configuration parameters can set up:

- The 200ppm target threshold. This value shall be 200ppm or higher.
- The filtering number n (see section above), It shall be between 0 and 8. Default is 7 which is similar to an averaging filter of 128 samples. A higher value will be more robust against noise. A lower value will track temperature variation more faster.

In order to prevent the host application domain from going into power down mode (power down mode not supported with FRO32K as clock source), the fwkSrvLowPowerConstraintCallbacks functions structure is registered to the Framework service on host application core from fwk_platform_lowpower.c file, PLATFORM_LowPowerInit() function. The NBU code applies a

low power Deep Sleep constraint to the application core. This constraint is released when the NBU firmware has no activity to do and re-applied when a new activity starts.

Lowpower impact

Power impact during active mode: In monitoring mode (this should be 99.9% of the time if temperature does not vary), the FRO32KHz frequency measurements are performed during a Radio activity so it does not increase the active current as the sources clocks are already active. Also, it does not increase the active time as the measurement takes less time than an advertising event or connection event so no impact on power consumption.

The main power impact will be in convergence mode. In this case, measurements/calibrations are done in loop until the monitoring mode is reached (frequency error less than 200ppm). This could happen:

- During power ON reset,
- When temperature varies: The frequency will deviate from 32768Hz and FRO32K trimming register correction will need to be updated for that,
- When no measurement has been done during some time as we cannot predict if the FRO has drifted, so we discard older values and start convergence mode.

When FRO32K frequency needs to be adjusted, the NBU core will wake-up the main power domain and will update the FRO32K trimming register.

Power impact during low power mode: The power consumption in low power mode will increase slightly due to running FRO32K compared to XTAL32K. The power consumption of FRO32K typically consumes 350nA while it is only 100nA with XTAL32K. Refer to the product datasheet for the exact numbers.

Chapter 4

RTOS

4.1 FreeRTOS

4.1.1 FreeRTOS kernel

Open source RTOS kernel for small devices.

FreeRTOS kernel for MCUXpresso SDK Readme

FreeRTOS kernel for MCUXpresso SDK ChangeLog

FreeRTOS kernel Readme

4.1.2 FreeRTOS drivers

This is set of NXP provided FreeRTOS reentrant bus drivers.

4.1.3 backoffalgorithm

Algorithm for calculating exponential backoff with jitter for network retry attempts.

Readme

4.1.4 corehttp

C language HTTP client library designed for embedded platforms.

4.1.5 corejson

JSON parser.

Readme

4.1.6 coremqtt

MQTT publish/subscribe messaging library.

4.1.7 corepkcs11

PKCS #11 key management library.

Readme

4.1.8 freertos-plus-tcp

Open source RTOS FreeRTOS Plus TCP.

Readme