



MCUXpresso SDK Documentation

Release 25.09.00-pvw1



NXP
Jul 17, 2025



Table of contents

1	Middleware	3
1.1	Motor Control	3
1.1.1	FreeMASTER	3
1.2	MultiCore	40
1.2.1	Multicore SDK	40
1.3	Wireless	135
1.3.1	NXP Wireless Framework and Stacks	135
2	RTOS	1501
2.1	FreeRTOS	1501
2.1.1	FreeRTOS kernel	1501
2.1.2	FreeRTOS drivers	1507
2.1.3	backoffalgorithm	1507
2.1.4	corehttp	1510
2.1.5	corejson	1512
2.1.6	coremqtt	1515
2.1.7	coremqtt-agent	1518
2.1.8	corepkcs11	1522
2.1.9	freertos-plus-tcp	1525

This documentation contains information specific to the imx95lp4xevk15 board.

Chapter 1

Middleware

1.1 Motor Control

1.1.1 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_pd_bdm** 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 applications 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_QUEUE_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-language 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_UINTn	Unsigned integer type of size <i>n</i> bits (n=8,16,32,64)
FMSTR_TSA_SINTn	Signed integer type of size <i>n</i> bits (n=8,16,32,64)
FMSTR_TSA_FRACn	Fractional number of size <i>n</i> bits (n=16,32,64).
FMSTR_TSA_FRAC_Q(<i>m,n</i>)	Signed fractional number in general Q form (m+n+1 total bits)
FMSTR_TSA_FRAC_UQ(<i>m,n</i>)	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)
FM-STR_TSA_USERTYPE(<i>name</i>)	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 *fmrstr:* 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
<i>FM-STR_ADDR</i> 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.	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.
<i>FM-STR_SIZE</i> It is required that this type is unsigned and at least 16 bits wide integer.	Data type used to hold the memory block size.
<i>FM-STR_BOOL</i> This type is used only in zero/non-zero conditions in the driver code.	Data type used as a general boolean type.
<i>FM-STR_APPCM</i> Generally, this is an unsigned 8-bit value.	Data type used to hold the Application Command code.
<i>FM-STR_APPCM</i> Generally, this is an unsigned 8-bit value.	Data type used to create the Application Command data buffer.
<i>FM-STR_APPCM</i> Generally, this is an unsigned 8-bit value.	Data type used to hold the Application Command result code.

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.

<i>FM-STR_TSA_TII</i>	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 <i>FM-STR_SIZE</i> .
<i>FM-STR_TSA_TS</i>	Data type used to hold a memory block size, as used in the TSA descriptors. By default, this is defined as <i>FM-STR_SIZE</i> .

Public Pipes types The table describes the data types used by the FreeMASTER Pipes API:

<i>FM-STR_HPIPE</i>	Pipe handle that identifies the open-pipe object. Generally, this is a pointer to a void type.
<i>FM-STR_PIPE_PC</i>	Integer type required to hold at least 7 bits of data. Generally, this is an unsigned 8-bit or 16-bit type.
<i>FM-STR_PIPE_SI</i>	Integer type required to hold at least 16 bits of data. This is used to store the data buffer sizes.
<i>FM-STR_PPIPEF</i>	Pointer to the pipe handler function. See FM-STR_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.

<i>FMSTR_U8</i>	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.	
<i>FMSTR_U16</i>	Unsigned 16-bit integer.
<i>FMSTR_U32</i>	Unsigned 32-bit integer.
<i>FMSTR_S8</i>	Signed 8-bit integer.
<i>FMSTR_S16</i>	Signed 16-bit integer.
<i>FMSTR_S32</i>	Signed 32-bit integer.
<i>FMSTR_FLOAT</i>	4-byte standard IEEE floating-point type.
<i>FMSTR_FLAGS</i>	Data type forming a union with a structure of flag bit-fields.
<i>FMSTR_SIZE8</i>	Data type holding a general size value, at least 8 bits wide.
<i>FMSTR_INDEX</i>	General for-loop index. Must be signed, at least 16 bits wide.
<i>FMSTR_BCHR</i>	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.	
<i>FMSTR_BPTR</i>	A pointer to the communication buffer (an array of <i>FMSTR_BCHR</i>).

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.

1.2 MultiCore

1.2.1 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 25.06.00.

This software package contains components for efficient work with multicore devices as well as for the multiprocessor communication.

What is new

- eRPC [CHANGELOG](#)
- RPPMsg-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
- 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

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	<MCUXpressoSDK_install_dir>/middleware/multicore/
Documentation	<MCSDK_dir>/mcuxsdk-doc/
Embedded Remote Procedure Call component	<MCSDK_dir>/erpc/
Multicore Manager component	<MCSDK_dir>/mcmgr/
RPMMsg-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 (RPMMsg-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_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

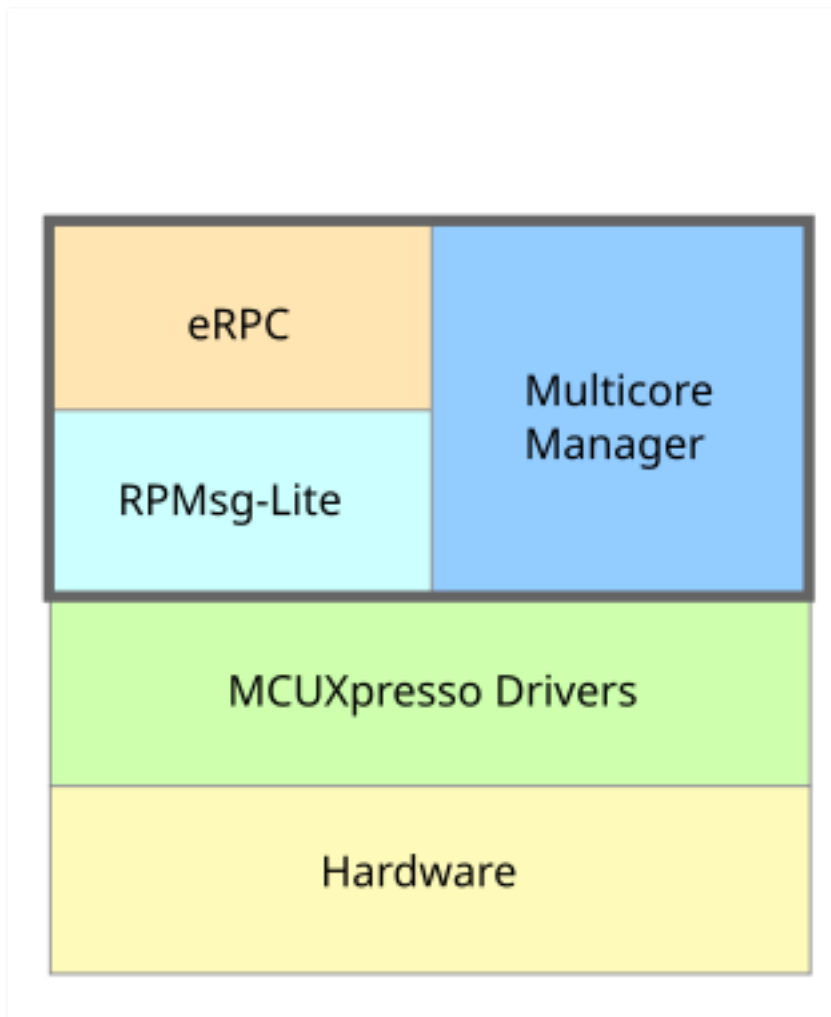
the <MCUXpressoSDK_install_dir>/examples/multiprocessor_examples subdirectories.

- 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.

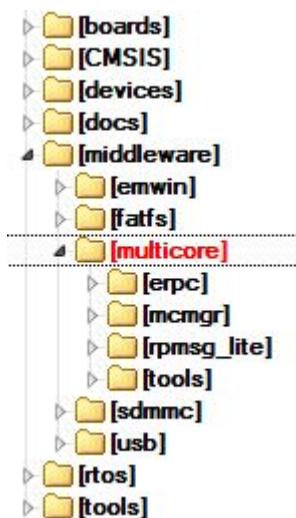


All the MCSDK-related files are located in `<MCUXpressoSDK_install_dir>/middleware/multicore` folder.

For supported toolchain versions, see the *Multicore SDK v25.06.00 Release Notes* (document MCS-DKRN). For the latest version of this and other MCSDK documents, visit www.nxp.com.

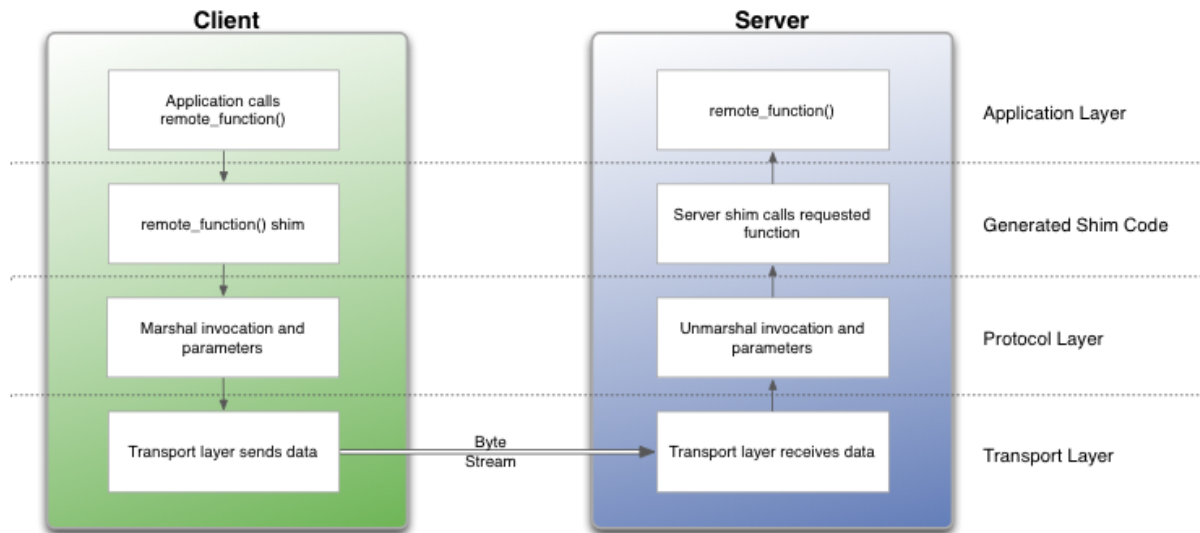
Multicore SDK (MCSDK) components The MCSDK consists of the following software components:

- **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 unmarshals 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 projects	example	<code><MCUXpressoSDK_install_dir>/examples/multicore_examples/<application_name>/</code>
Multiprocessor projects	example	<code><MCUXpressoSDK_install_dir>/examples/multiprocessor_examples/<application_name>/</code>

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](#).

[25.06.00]

- 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
- RPPMsg-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
 - RPPMsg-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
 - RPPMsg-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
 - RPPMsg-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
 - RPPMsg-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
 - RPPMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPSMsg-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
 - RPMMsg-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 / RPMMsg 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 / RPMMsg 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.

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.

RPMMSG-Lite This documentation describes the RPMMsg-Lite component, which is a lightweight implementation of the Remote Processor Messaging (RPMMsg) protocol. The RPMMsg protocol defines a standardized binary interface used to communicate between multiple cores in a heterogeneous multicore system.

Compared to the RPMMsg implementation of the Open Asymmetric Multi Processing (OpenAMP) framework (<https://github.com/OpenAMP/open-amp>), the RPMMsg-Lite offers a code size reduction, API simplification, and improved modularity. On smaller Cortex-M0+ based systems, it is recommended to use RPMMsg-Lite.

The RPMMsg-Lite is an open-source component developed by NXP Semiconductors and released under the BSD-compatible license.

For Further documentation, please look at doxygen documentation at: <https://nxp-mcuxpresso.github.io/rpmsg-lite/>

Motivation to create RPMMsg-Lite There are multiple reasons why RPMMsg-Lite was developed. One reason is the need for the small footprint of the RPMMsg protocol-compatible communication component, another reason is the simplification of extensive API of OpenAMP RPMMsg implementation.

RPMMsg 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.

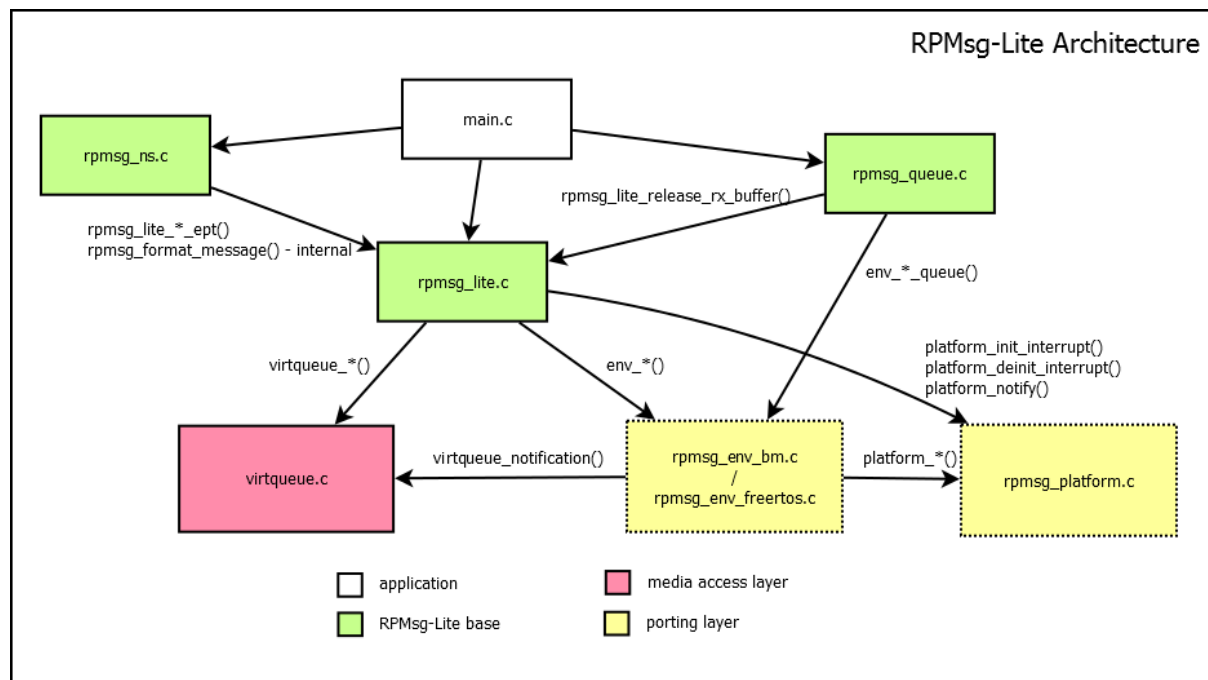
Small MCU-based systems often do not implement dynamic memory allocation. The creation of static API in RPMMsg-Lite enables another reduction of resource usage. Not only does the dynamic allocation adds another 5 KB of code size, but also communication is slower and less deterministic, which is a property introduced by dynamic memory. The following table shows some rough comparison data between the OpenAMP RPMMsg implementation and new RPMMsg-Lite implementation:

Component / Configuration	Flash [B]	RAM [B]
OpenAMP RPMMsg / Release (reference)	5547	456 + dynamic
RPMMsg-Lite / Dynamic API, Release	3462	56 + dynamic
Relative Difference [%]	~62.4%	~12.3%
RPMMsg-Lite / Static API (no malloc), Release	2926	352
Relative Difference [%]	~52.7%	~77.2%

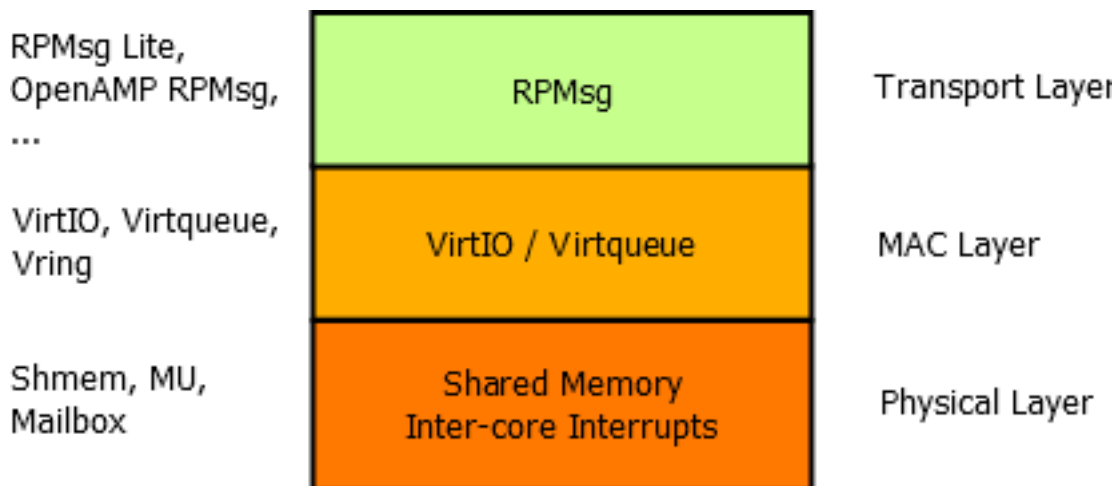
Implementation The implementation of RPMMsg-Lite can be divided into three sub-components, from which two are optional. The core component is situated in `rpmsg_lite.c`. Two optional components are used to implement a blocking receive API (in `rpmsg_queue.c`) and dynamic “named” endpoint creation and 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 `/rpmmsg_lite/lib/include` directory to the include path and in the application, include either the `rpmmsg_lite.h` header file, or optionally also include the `rpmmsg_queue.h` and/or `rpmmsg_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, `rpmmsg_env_myrtos.c`) and to include it in the project build.

The initialization of the stack is done by calling the `rpmmsg_lite_master_init()` on the master side and the `rpmmsg_lite_remote_init()` on the remote side. This initialization function must be called prior to any RPMMsg-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 `rpmmsg_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 `rpmmsg_queue_create()` function must be called before calling `rpmmsg_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 `rpmmsg_queue_rx_cb()`. Then, it is possible to use `rpmmsg_queue_receive()` function to listen on a queue object for incoming messages. The `rpmmsg_lite_send()` function is used to send messages to the other side.

The RPMMsg-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.

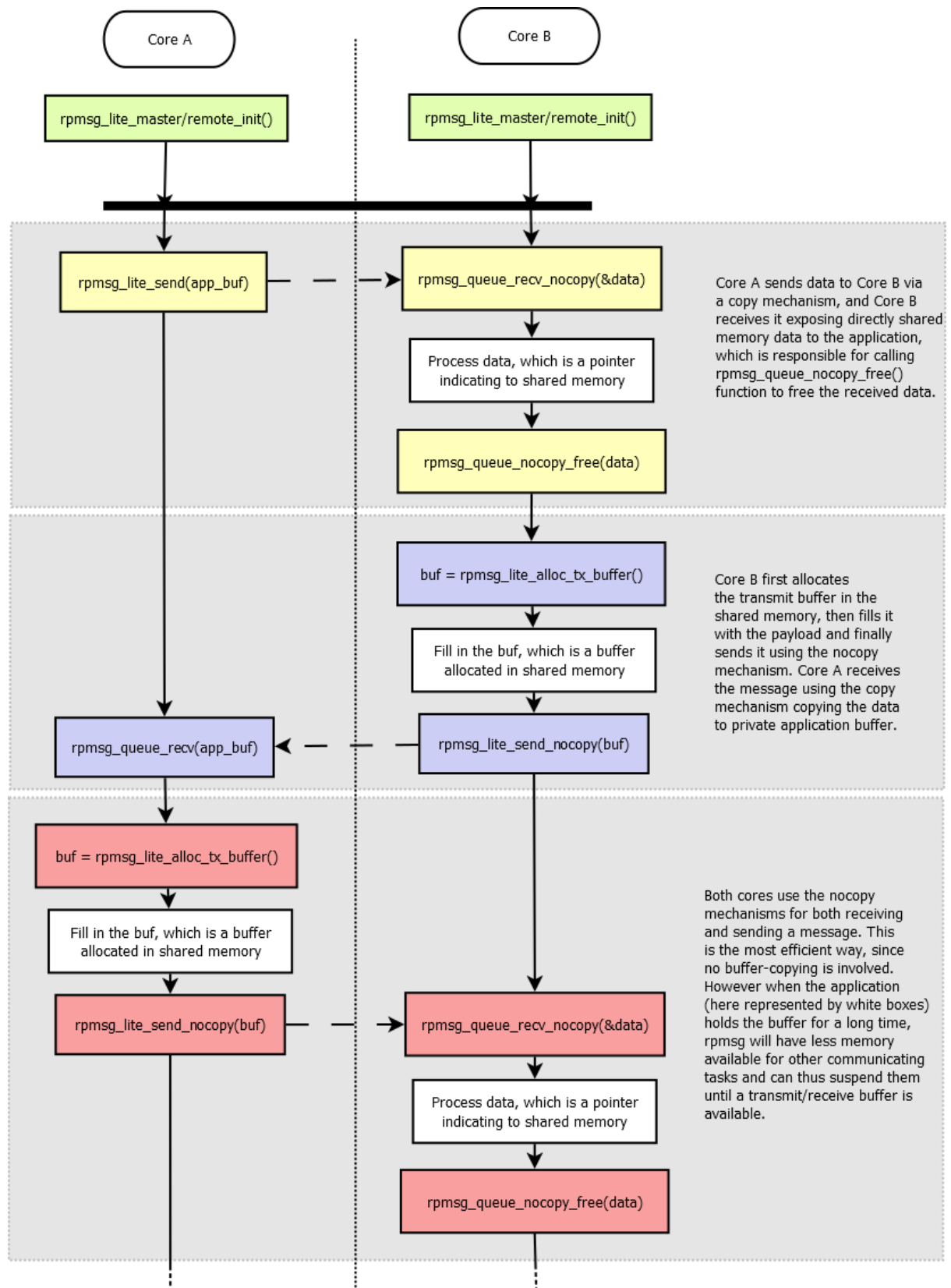
no-copy-send mechanism: This mechanism allows sending messages without the cost for copying data from the application buffer to the RPMMsg/virtio buffer in the shared memory. The sequence of no-copy sending steps to be performed is as follows:

- Call the `rpmmsg_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 `rpmmsg_lite_alloc_tx_buffer()` size output parameter).
- Call the `rpmmsg_lite_send_nocopy()` function to send the message to the destination endpoint. Consider the cache functionality and the virtio buffer alignment. See the `rpmmsg_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 `rpmmsg_queue_rcv_nocopy()` function to get the virtio buffer pointer to the received data.
- Read received data directly from the shared memory.
- Call the `rpmmsg_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 RPMMsg-Lite objects he has created in case of deinitialization. In order to do this, the function `rpmmsg_queue_destroy()` is used to destroy a queue, `rpmmsg_lite_destroy_ept()` is used to destroy an endpoint and finally, `rpmmsg_lite_deinit()` is used to deinitialize the RPMMsg-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. RPMMsg-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 RPSMsg_Lite multicore examples with 'rpsmsg_lite_' name prefix.

Another way of getting NXP MCUXpressoSDK RPSMsg_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 rpsmsg_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 RPSMsg_Lite examples use the 'rpsmsg_lite_' name prefix.

Notes

Environment layers implementation Several environment layers are provided in lib/rpsmsg_lite/porting/environment folder. Not all of them are fully tested however. Here is the list of environment layers that passed testing:

- rpsmsg_env_bm.c
- rpsmsg_env_freertos.c
- rpsmsg_env_xos.c
- rpsmsg_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 linker file is also a good practise, it is recommended to use linker files from MCUXpressoSDK 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 RPSMsg-Lite can be configured at the compile time. The default configuration is defined in the rpsmsg_default_config.h header file. This configuration can be customized by the user by including rpsmsg_config.h file with custom settings. The following table summarizes all possible RPSMsg-Lite configuration options.

Config- uration option	De- fault value	Usage
RL_MS_PE (1)		Delay in milliseconds used in non-blocking API functions for polling.
RL_BUFEE (496)		Size of the buffer payload, it must be equal to (240, 496, 1008, ...) $[2^n - 16]$
RL_BUFEE (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 <code>rpmsg_lite_send_nocopy()</code> and <code>rpmsg_lite_release_rx_buffer()</code> functions (enabled by <code>RL_API_HAS_ZEROCOPY</code> 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 <code>rpmsg_config.h</code> . 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_ASSERT	see <code>rpmsg</code>	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](#).

Unreleased

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.
- Baremetal 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](#)

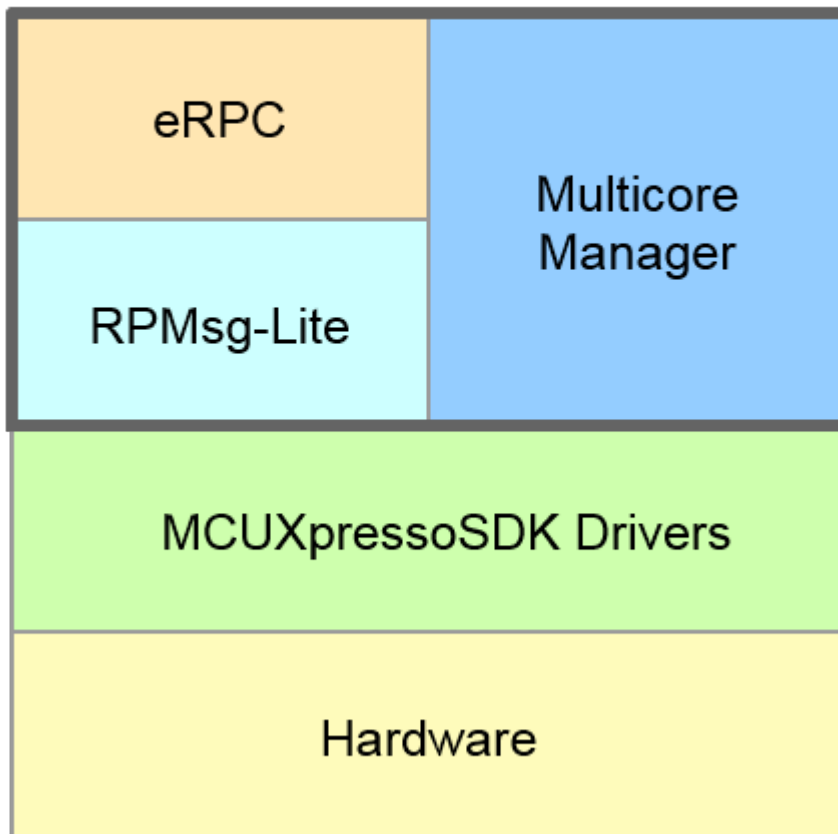
Visit [Multicore Manager - Documentation](#) to review details on the contents in this sub-repo.

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();

    /* 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.

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();
```

(continues on next page)

(continued from previous page)

```

/* Register the application event before starting the secondary core */
MCMGR_RegisterEvent(kMCMGR_RemoteApplicationEvent, RPSMsgRemoteReadyEventHandler, (void_)
↳)RPSMsgRemoteReadyEventData);

/* Boot secondary core application from the CORE1_BOOT_ADDRESS, pass rpsmsg_lite_base address_
↳as startup data, starting synchronously. */
MCMGR_StartCore(APP_SECONDARY_CORE, CORE1_BOOT_ADDRESS, (uint32_t)rpsmsg_lite_
↳base, kMCMGR_Start_Synchronous);

/* Wait until the secondary core application signals the rpsmsg remote has been initialized and is ready to_
↳communicate. */
while(APP_RPSMSG_READY_EVENT_DATA != RPSMsgRemoteReadyEventData[APP_SECONDARY_
↳CORE]) {};
.
.
.
}

```

Code for the secondary side:

```

#include "mcmgr.h"

#define APP_RPSMSG_READY_EVENT_DATA (1)

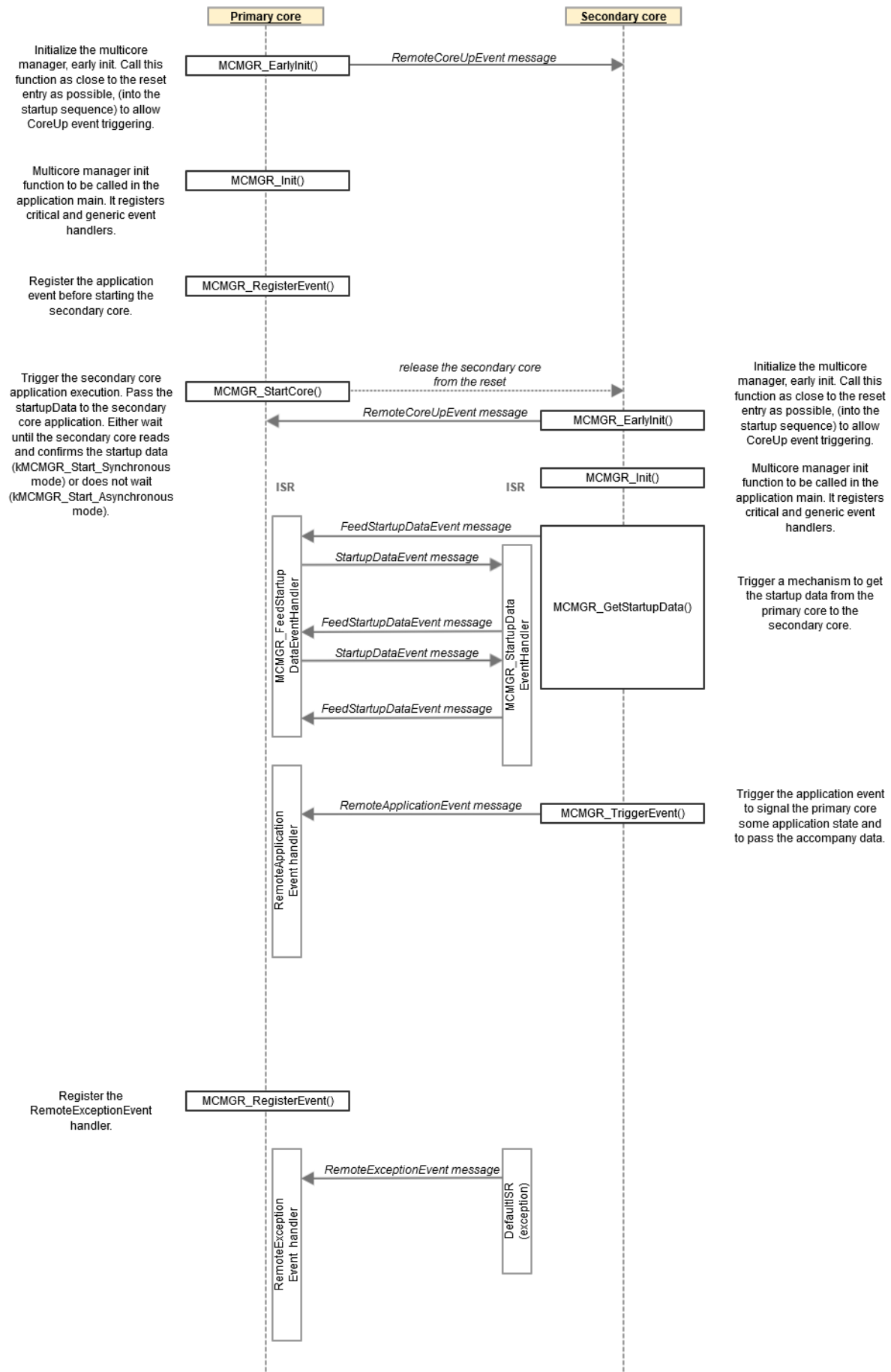
void main()
{
    /* Initialize MCMGR - low level multicore management library.
       Call this function as close to the reset entry as possible,
       (into the startup sequence) to allow CoreUp event triggering. */
    MCMGR_EarlyInit();

    /* 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_RPSMSG_
    ↳READY_EVENT_DATA */
    MCMGR_TriggerEvent(kMCMGR_Core0, kMCMGR_RemoteApplicationEvent, APP_RPSMSG_
    ↳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.



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](#).

Unreleased

Added

Fixed

- Added CX flag into CMakeLists.txt to allow c++ build compatibility.

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 than 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: [Porting-GuideTo_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);
    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 */
```

(continues on next page)

(continued from previous page)

```

    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
{
    /* 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 */
}

```

(continues on next page)

(continued from previous page)

```
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 be 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, ...**
- **Bison** - <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 configure 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 configure 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:

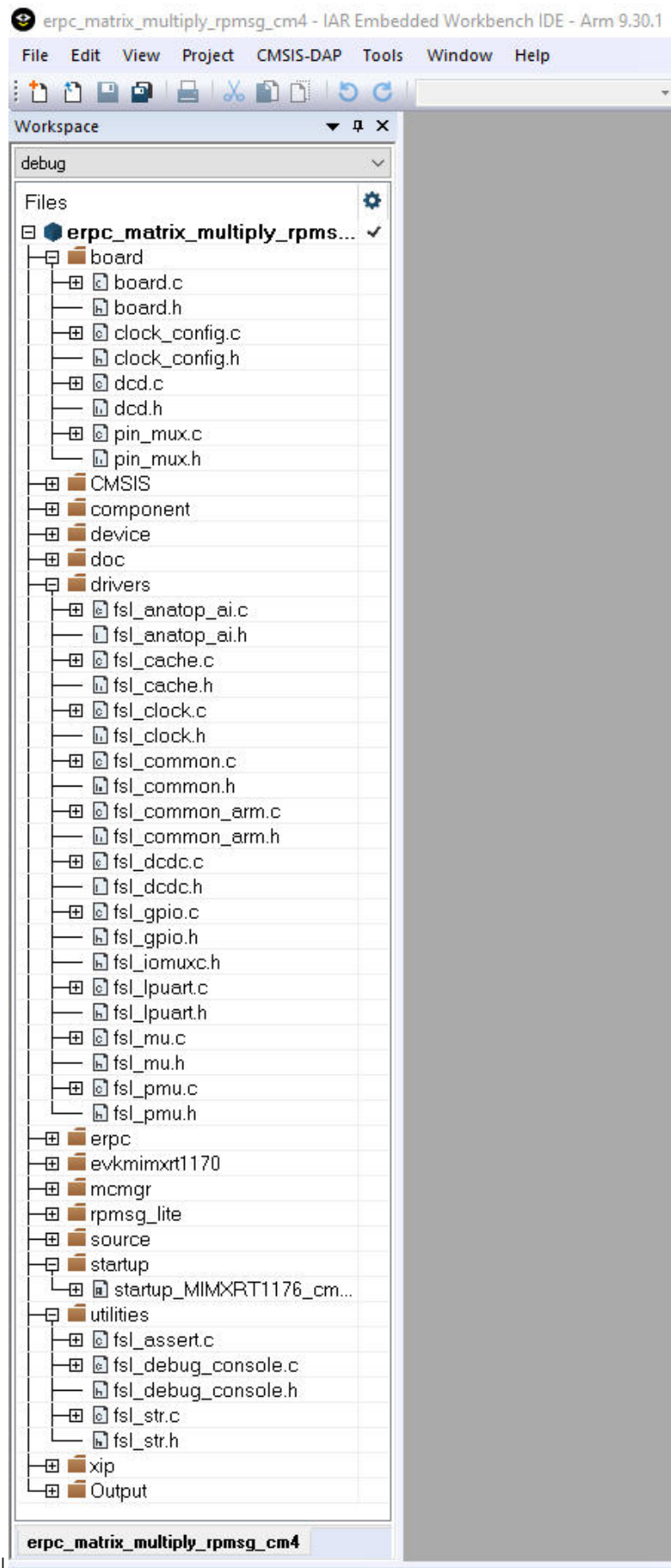
`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmmsg/cm4/iar`

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>/`





|

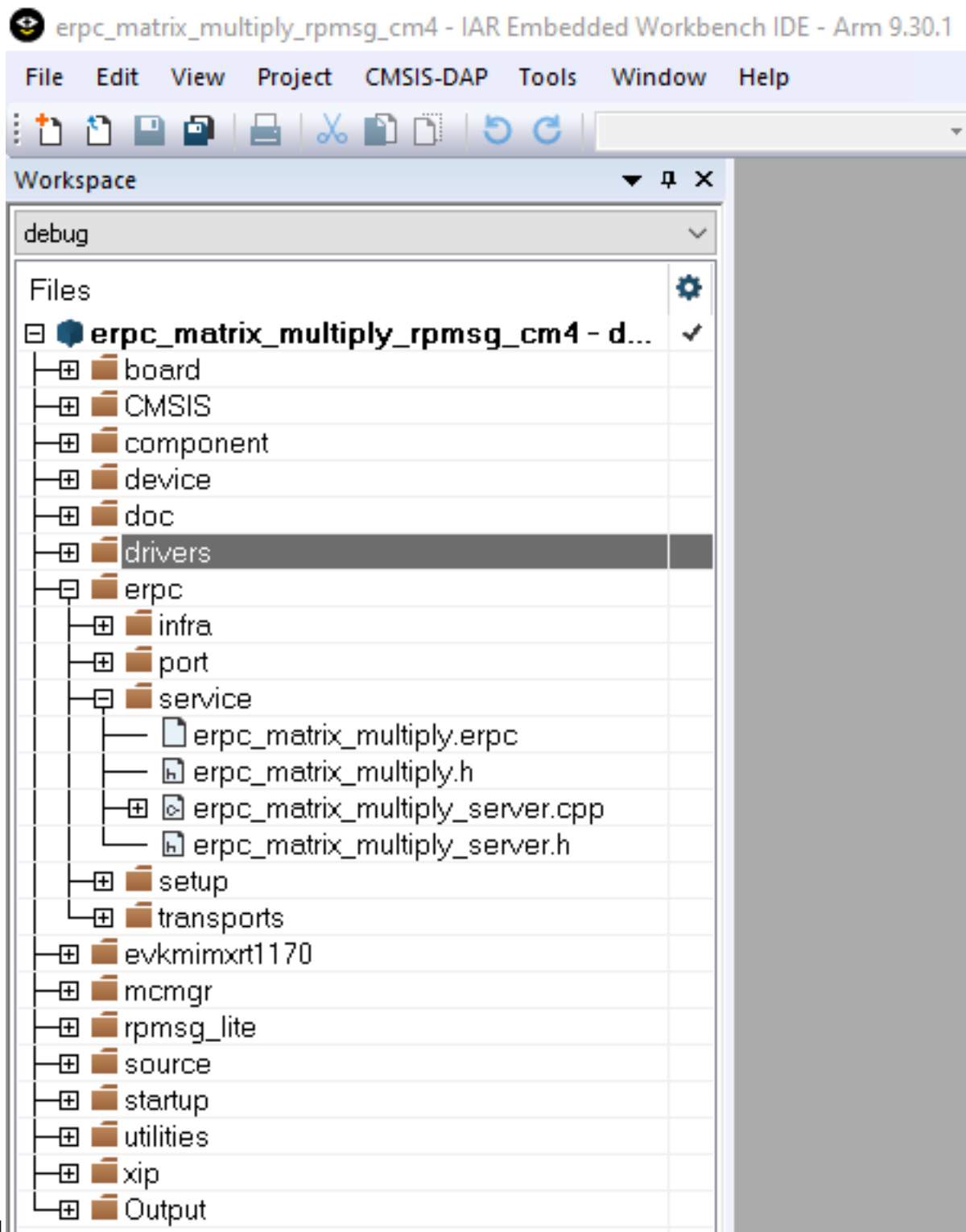
**Parent topic:** Multicore server application

**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/



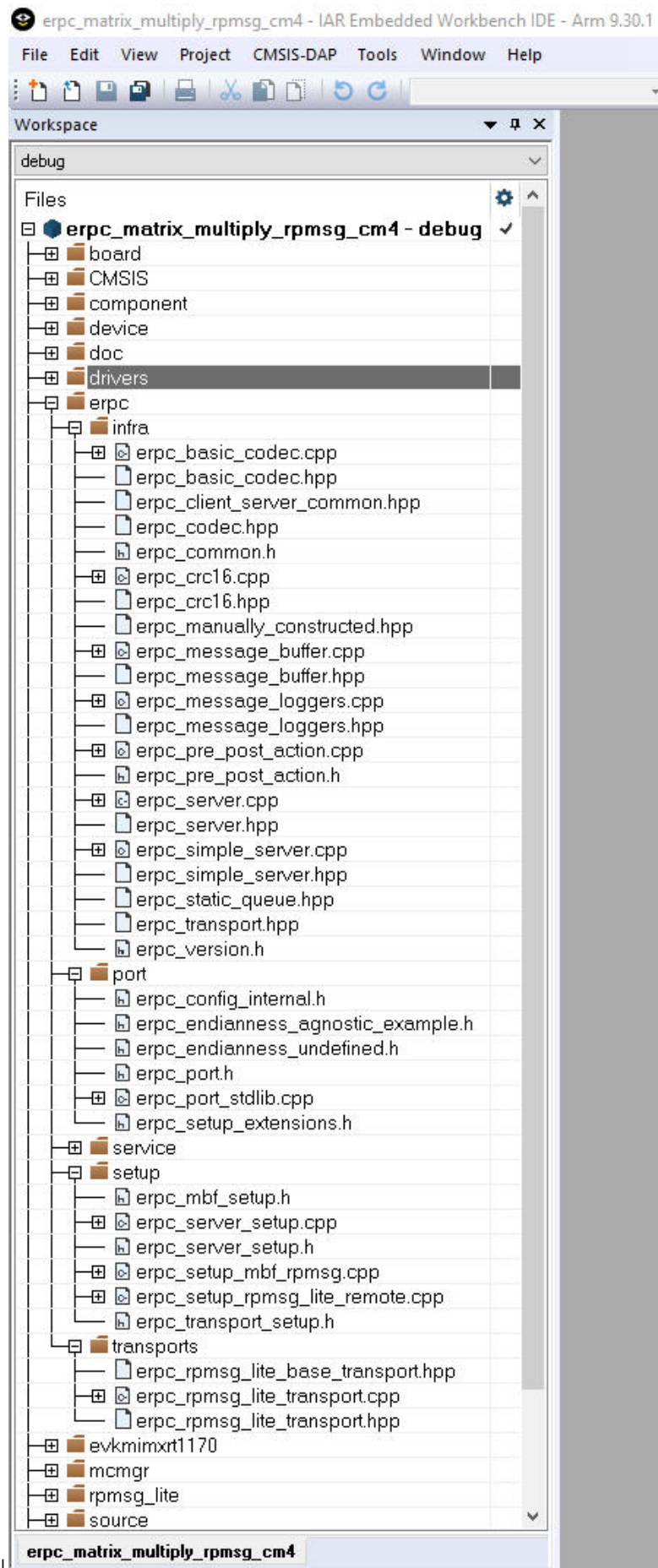
**Parent topic:** Multicore server application

**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 need 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 need 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 need 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.



|

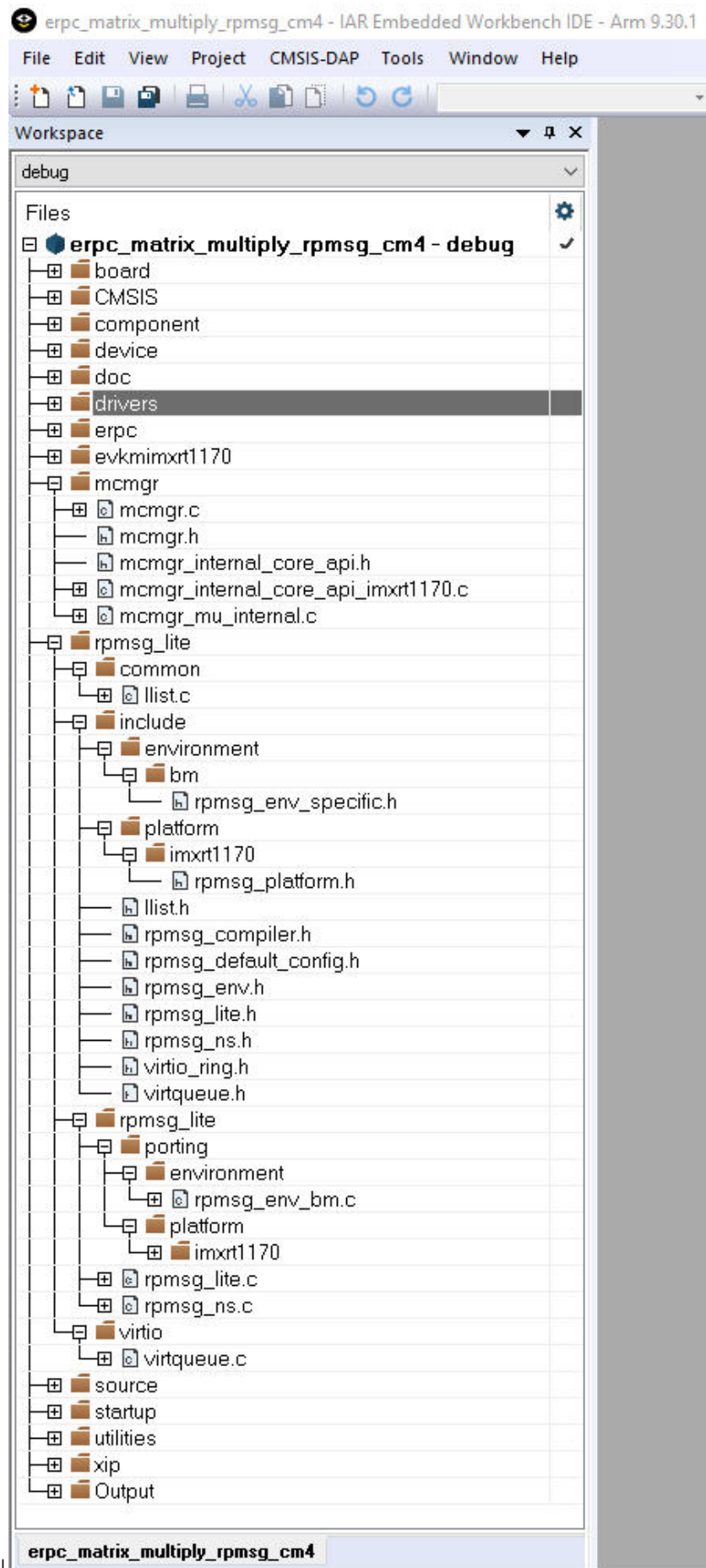
**Parent topic:** Multicore server application

**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/`



|

**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:

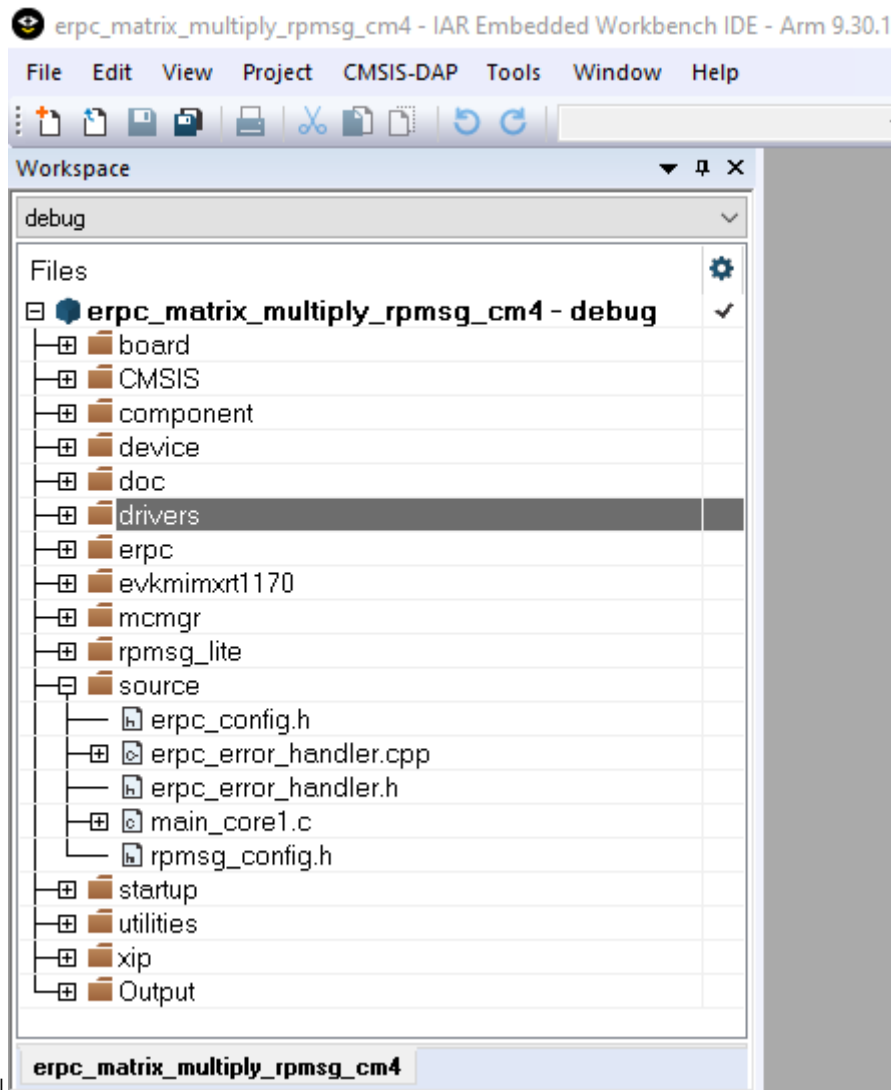
- The **main()** function contains the code for the target board and eRPC server initialization. After the initialization, the matrix multiply service is added and the eRPC server waits for client's requests in the while loop.
- The **erpcMatrixMultiply()** function is the user implementation of the 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()
{
 ...
 /* RPSMsg-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](#)

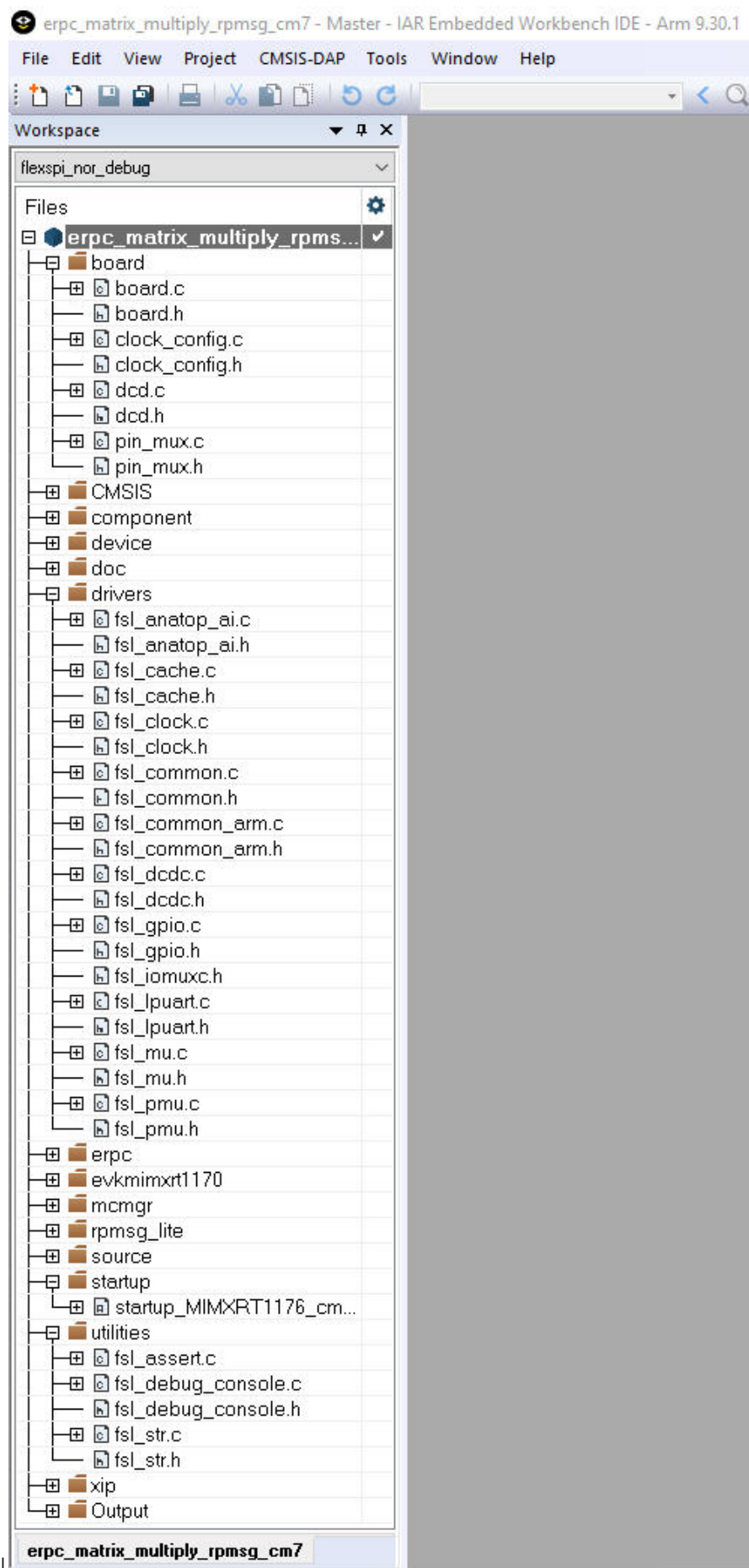
**Multicore client application** The “Matrix multiply” eRPC client project is located in the following folder:

`<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>/`



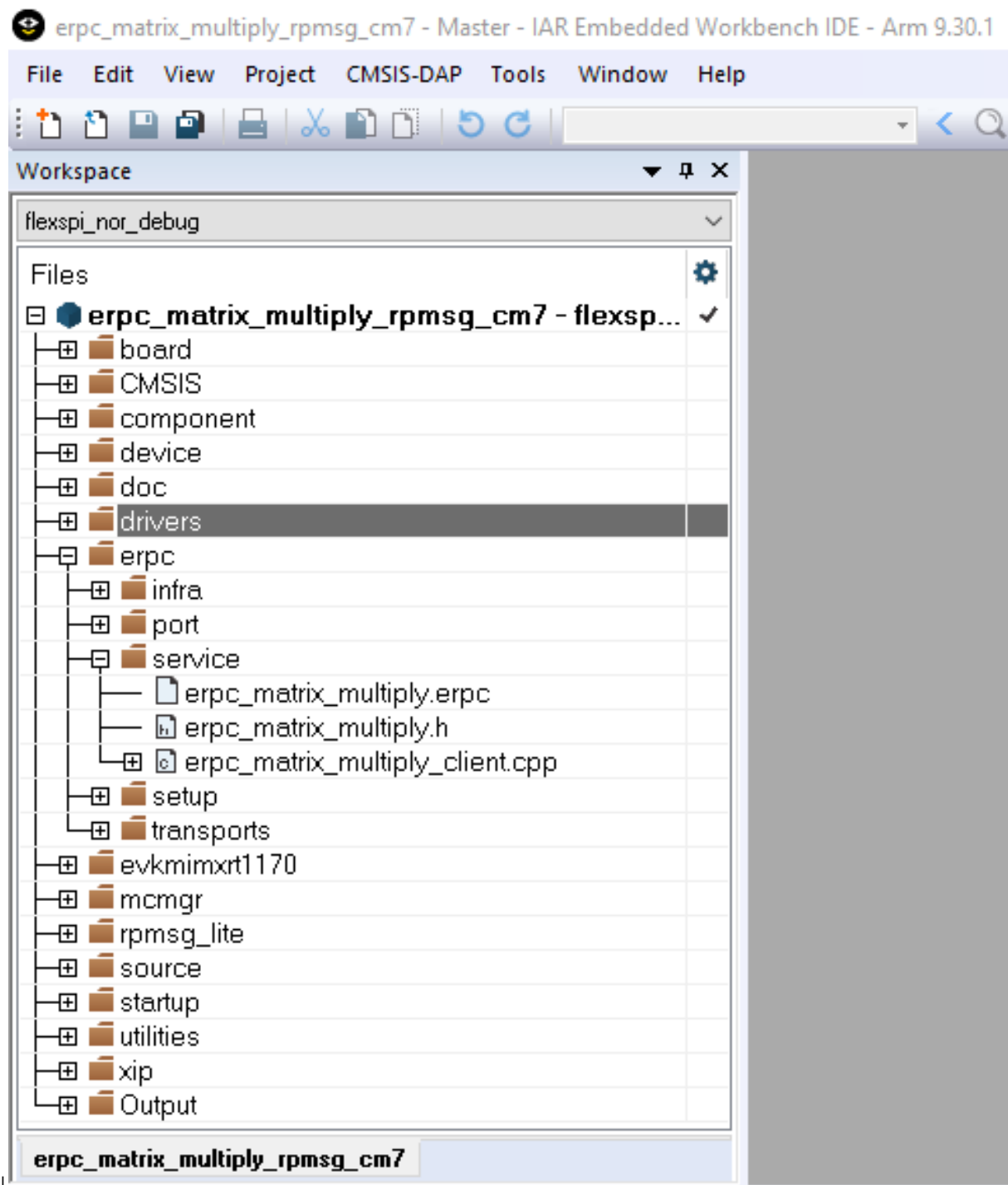
|

**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.

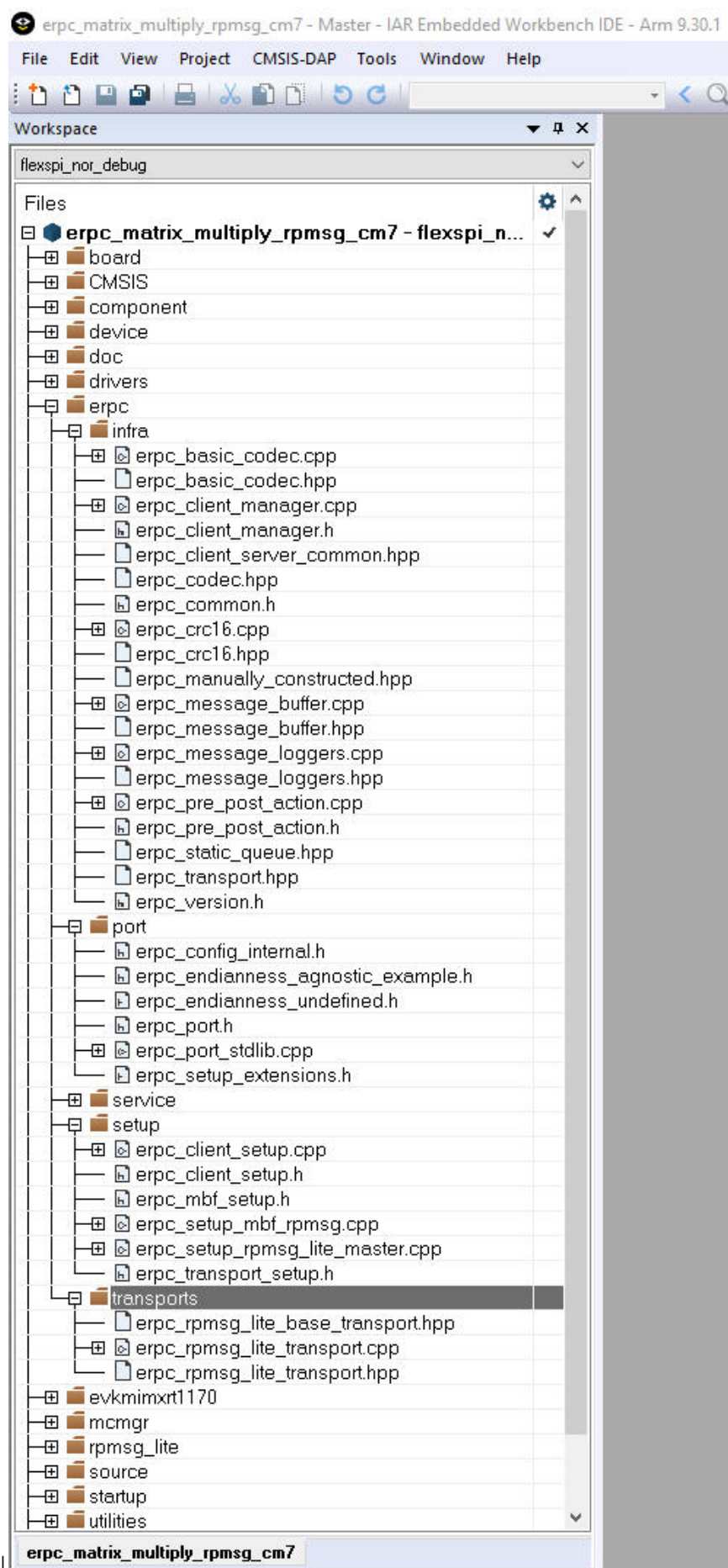
- `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
- `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
- `erpc_config_internal.h` internal eRPC configuration file.

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.



|

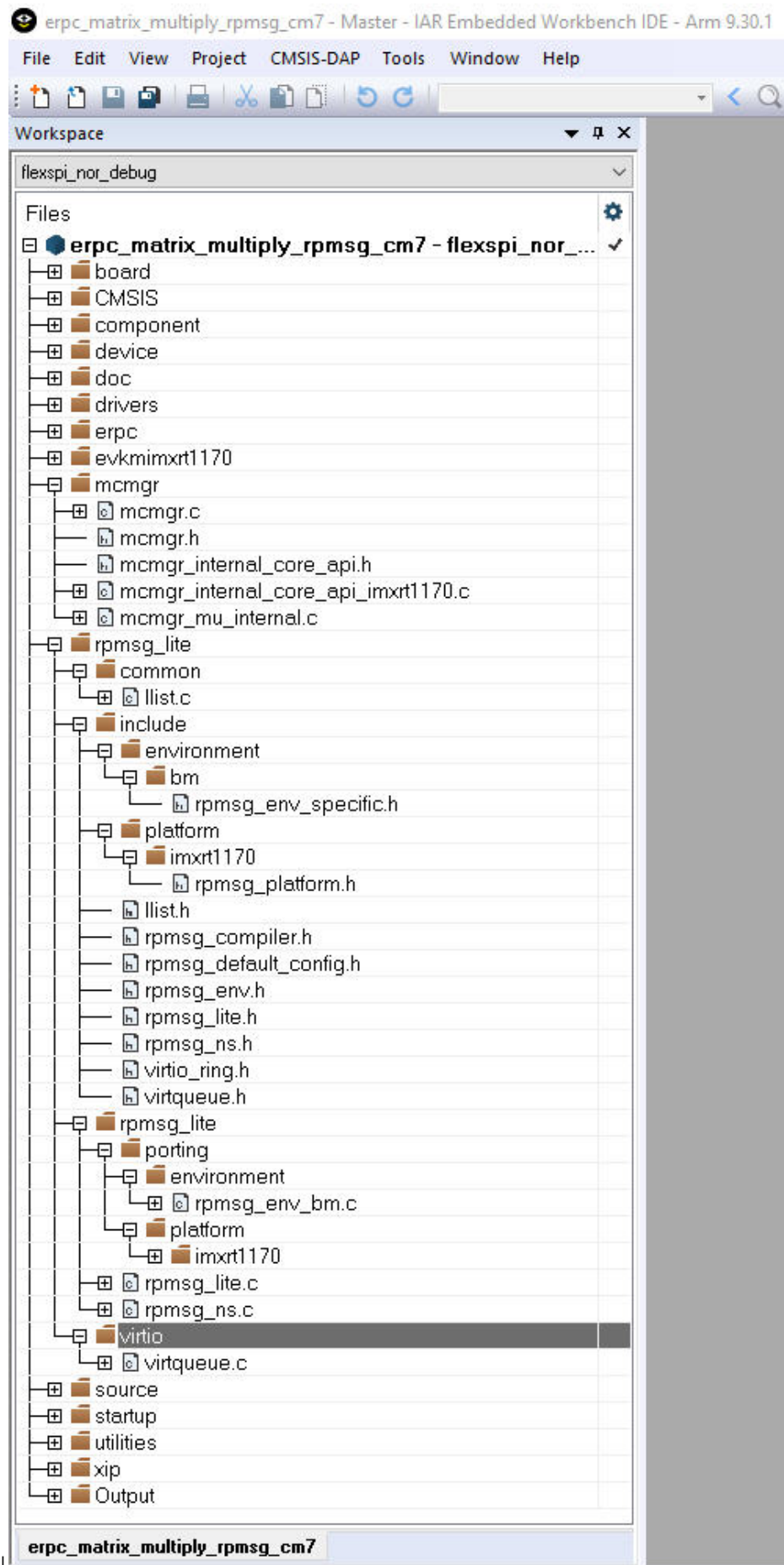
**Parent topic:**Multicore client application

**Client multicore infrastructure files** Because of the RPSMsg-Lite (transport layer), it is also necessary to include RPSMsg-Lite related files, which are in the following folder:

*<MCUXpressoSDK\_install\_dir>/middleware/multicore/rpsmsg\_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/*





**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.

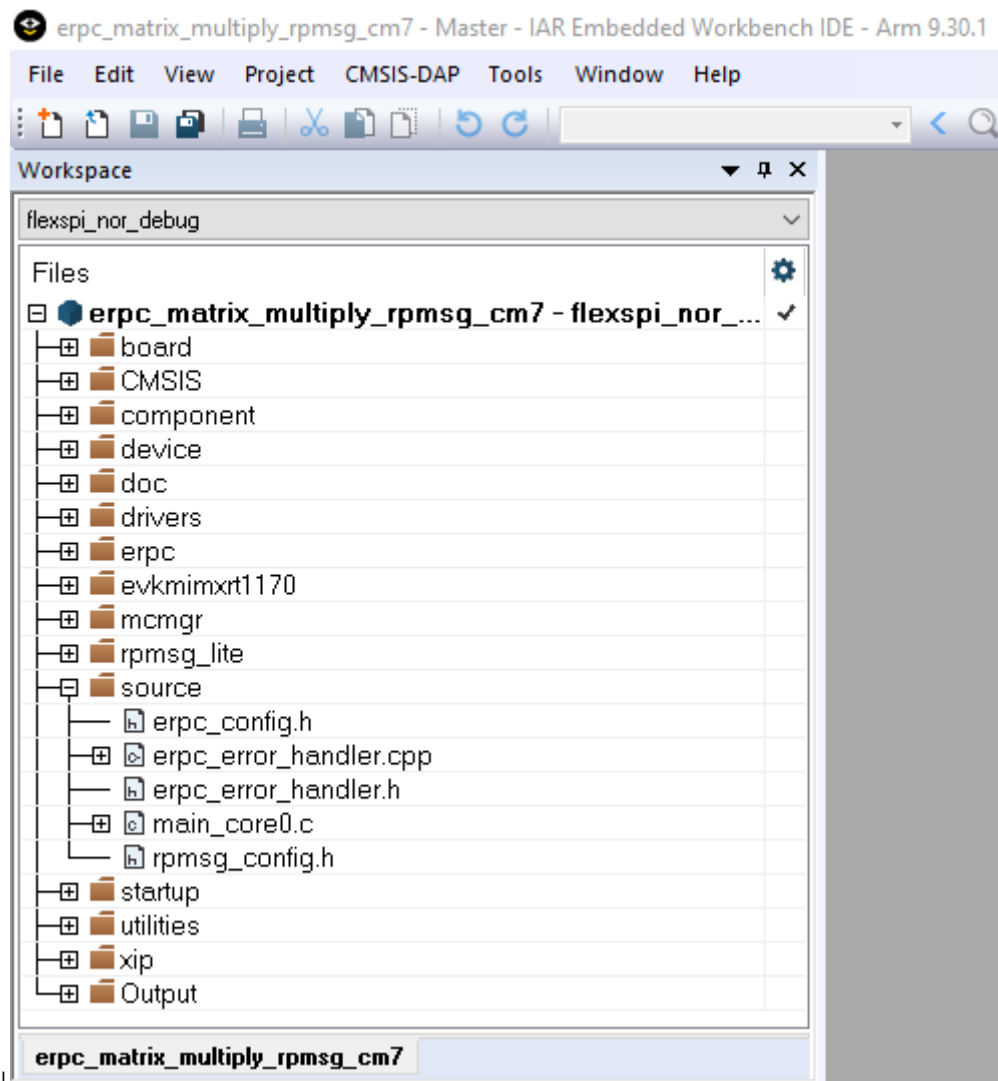
The matrix multiplication can be issued repeatedly, when pressing a software board button.

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 occurred 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/transport/erpc\_uart\_cmsis\_transport.hpp

<eRPC base directory>/erpc\_c/transport/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 RPSMsg-Lite). The RPSMsg-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:

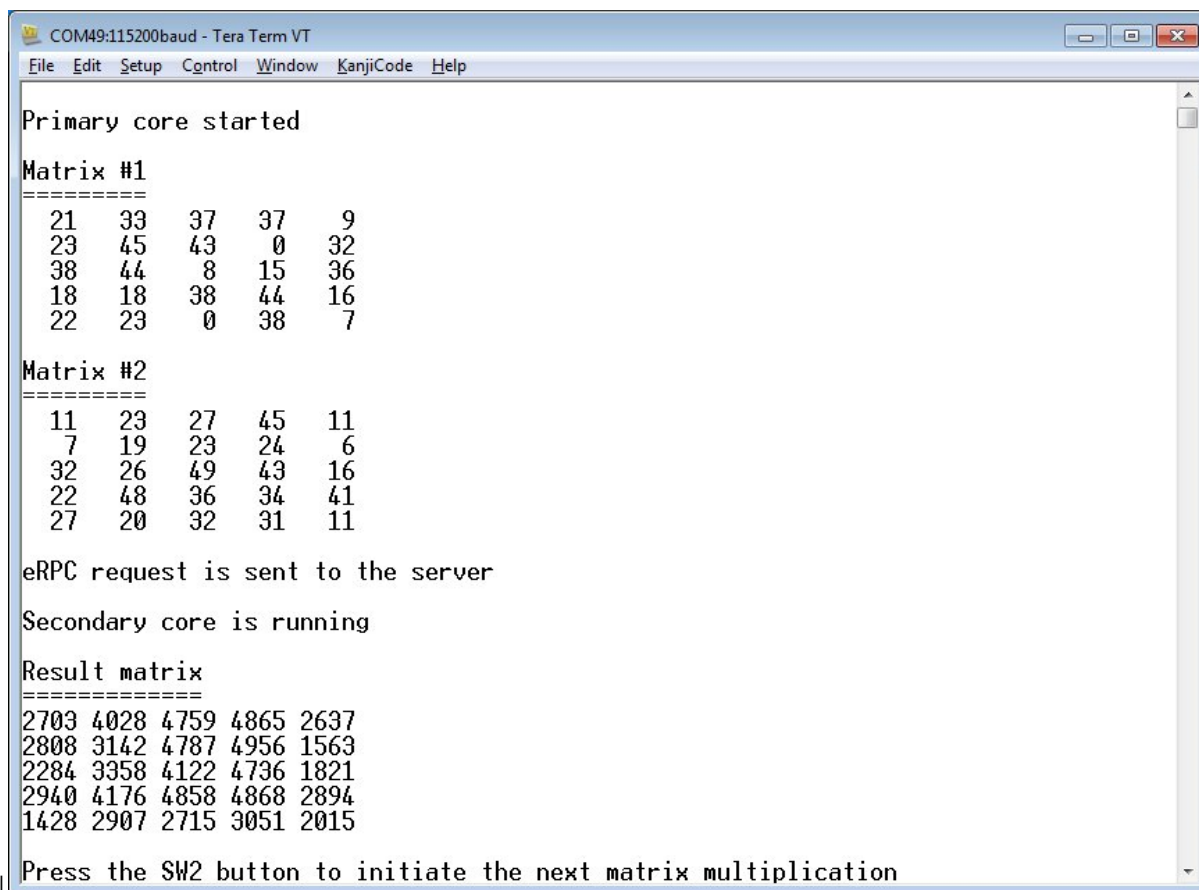
```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver
↳operations */
erpc_transport_t transport;
transport = erpc_transport_cmsis_uart_init((void *)&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 occurred 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:



```

COM49:115200baud - Tera Term VT
File Edit Setup Control Window KanjiCode Help

Primary core started

Matrix #1
=====
 21 33 37 37 9
 23 45 43 0 32
 38 44 8 15 36
 18 18 38 44 16
 22 23 0 38 7

Matrix #2
=====
 11 23 27 45 11
 7 19 23 24 6
 32 26 49 43 16
 22 48 36 34 41
 27 20 32 31 11

eRPC request is sent to the server

Secondary core is running

Result matrix
=====
2703 4028 4759 4865 2637
2808 3142 4787 4956 1563
2284 3358 4122 4736 1821
2940 4176 4858 4868 2894
1428 2907 2715 3051 2015

Press the SW2 button to initiate the next matrix multiplication

```

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 communicates with the eRPC server that runs on another 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/` | | RPLite  
source files | `<MCUXpressoSDK_install_dir>/middleware/multicore/rplite/`

**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 been 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/`

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/erpcgen/Linux_x86`  
 | 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/service` 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:

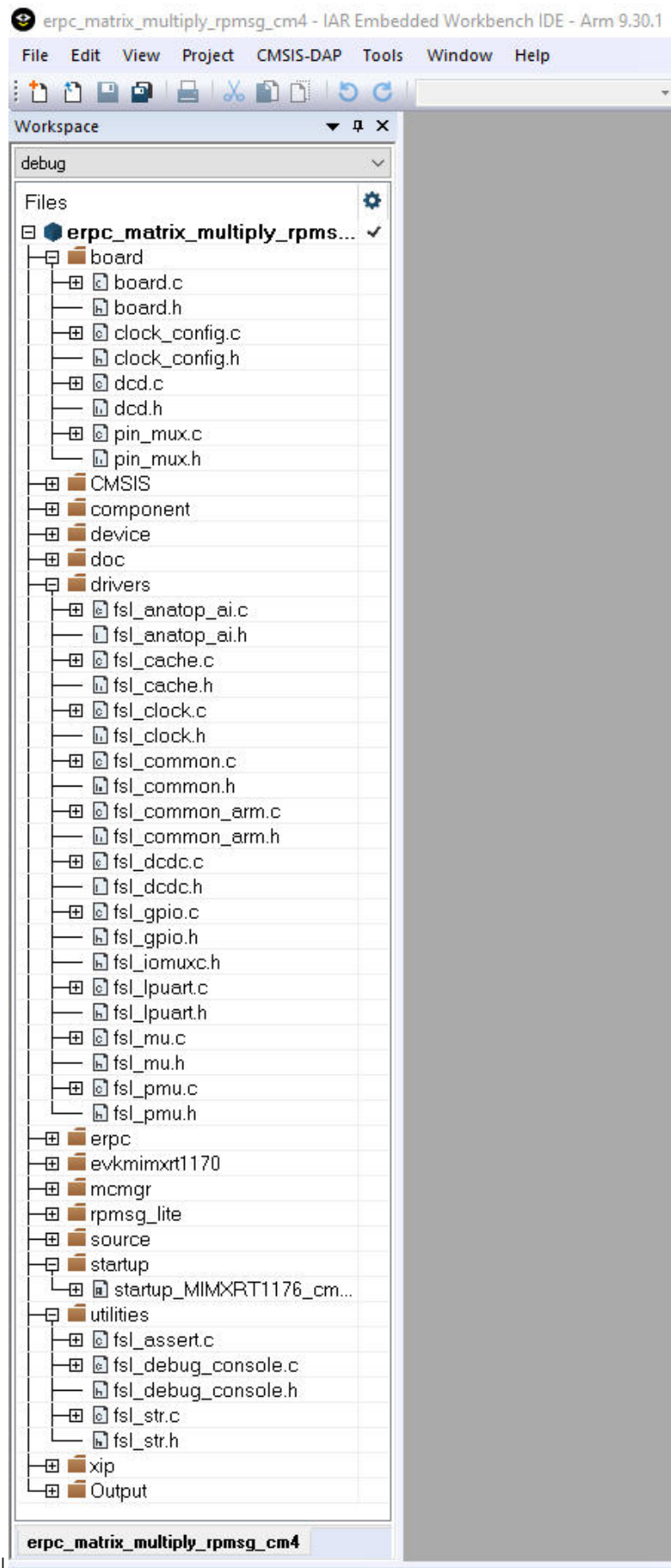
`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmmsg/cm4/iar/`

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>/`





|

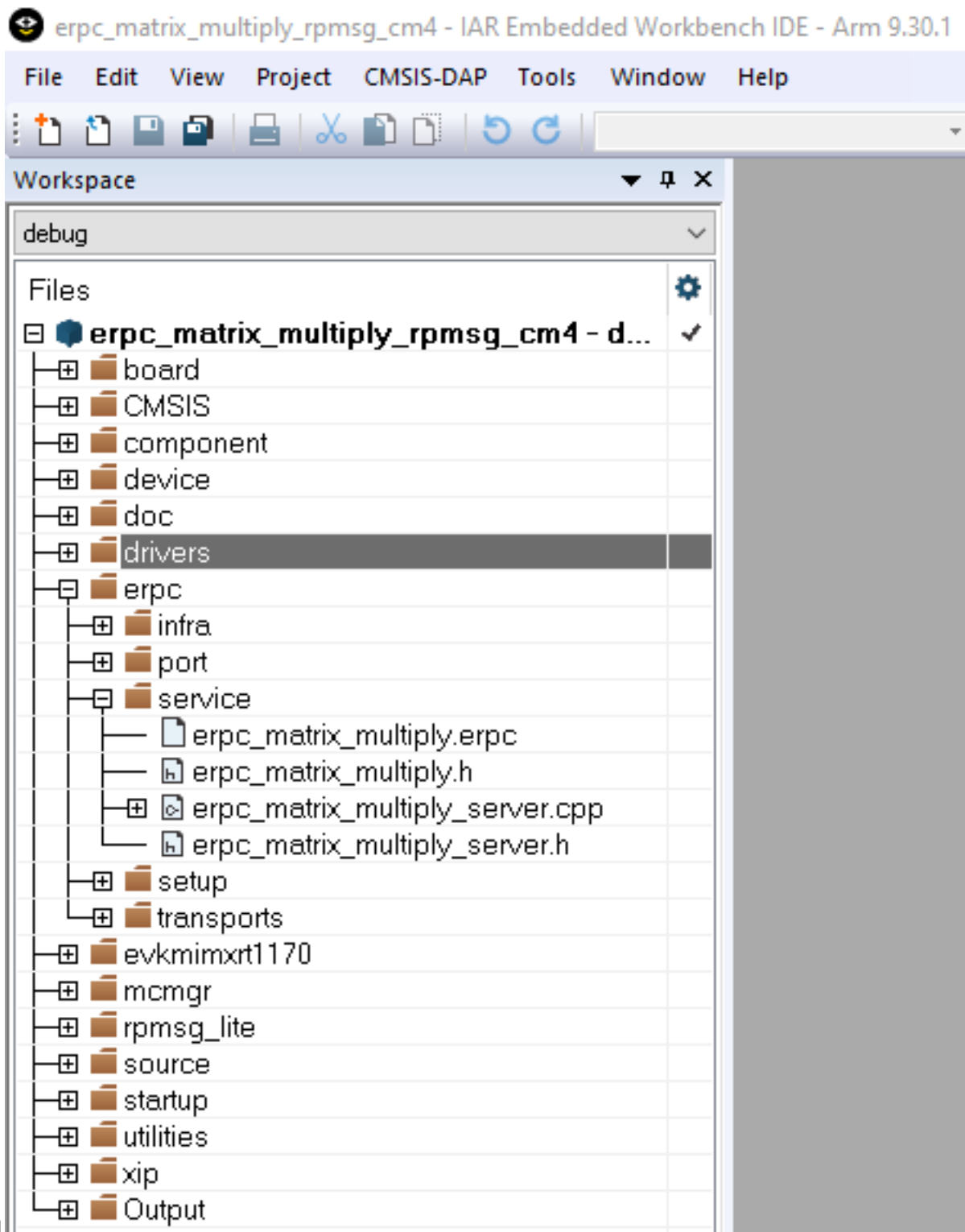
**Parent topic:**Multicore server application

**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/



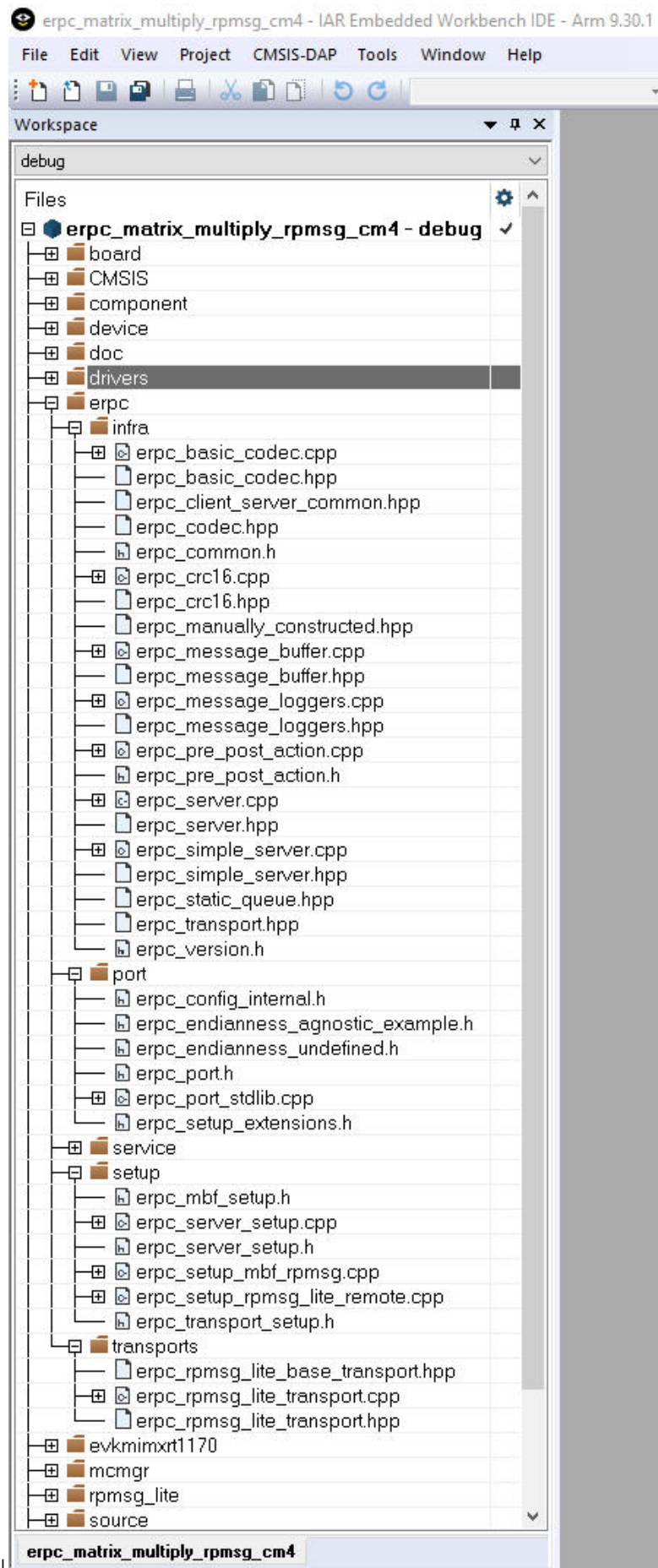
**Parent topic:** Multicore server application

**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 need 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 need 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 need 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.



|

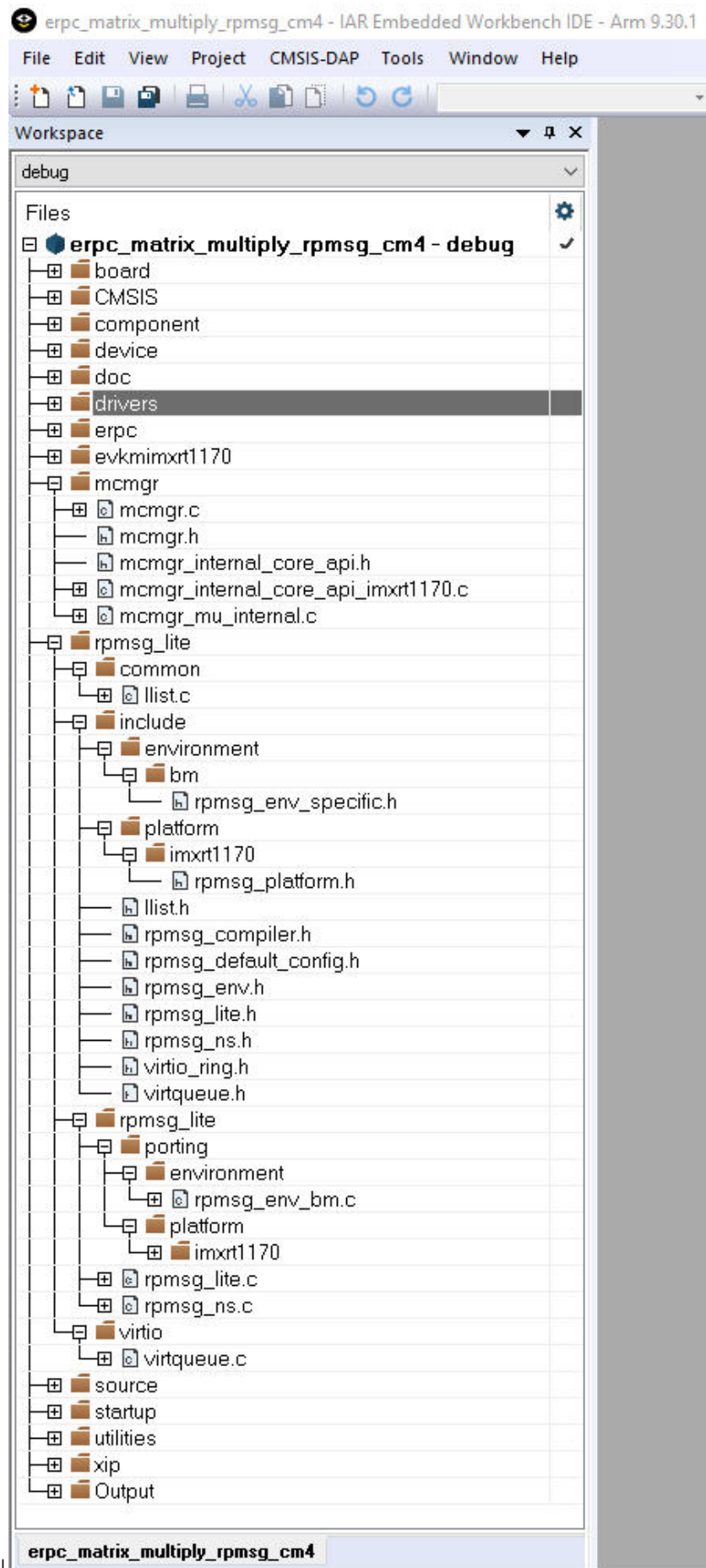
**Parent topic:** Multicore server application

**Server multicore infrastructure files** Because of the RPSMsg-Lite (transport layer), it is also necessary to include RPSMsg-Lite related files, which are in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/rpsmsg_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/`



|

**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:

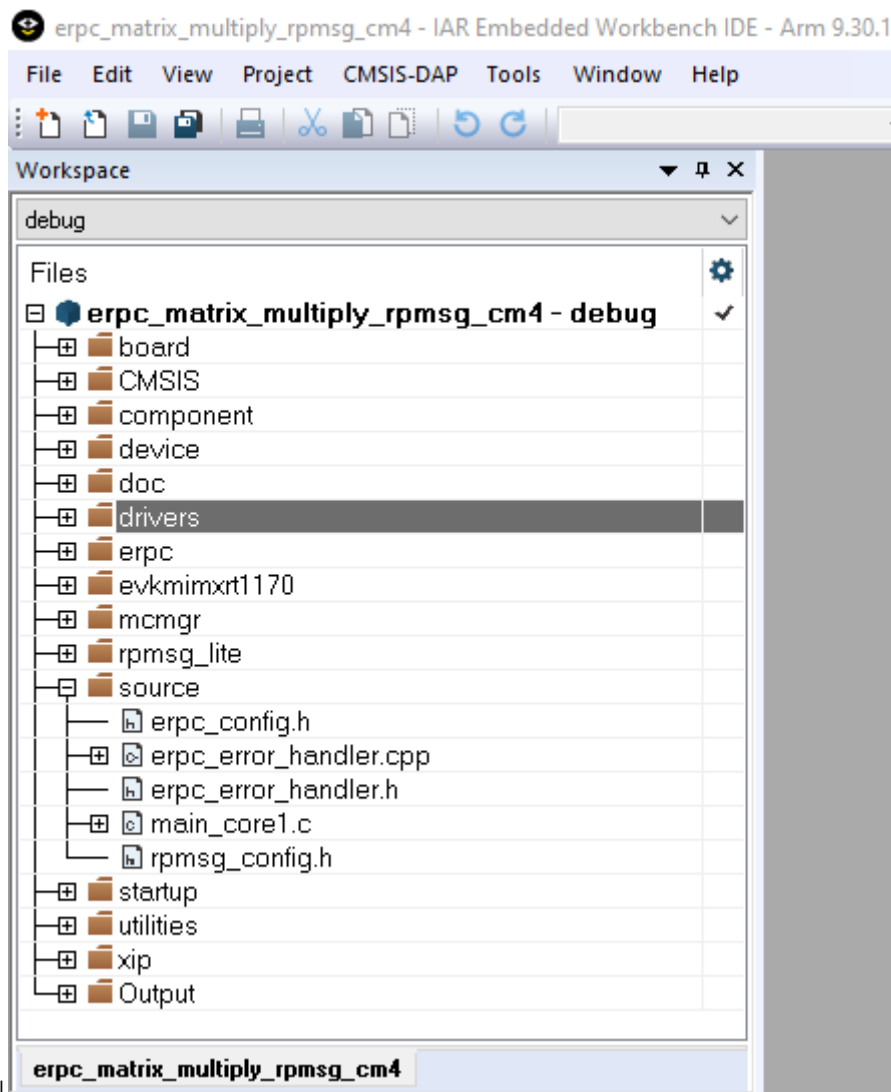
- The **main()** function contains the code for the target board and eRPC server initialization. After the initialization, the matrix multiply service is added and the eRPC server waits for client's requests in the while loop.
- The **erpcMatrixMultiply()** function is the user implementation of the 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()
{
 ...
 /* RPSMsg-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](#)

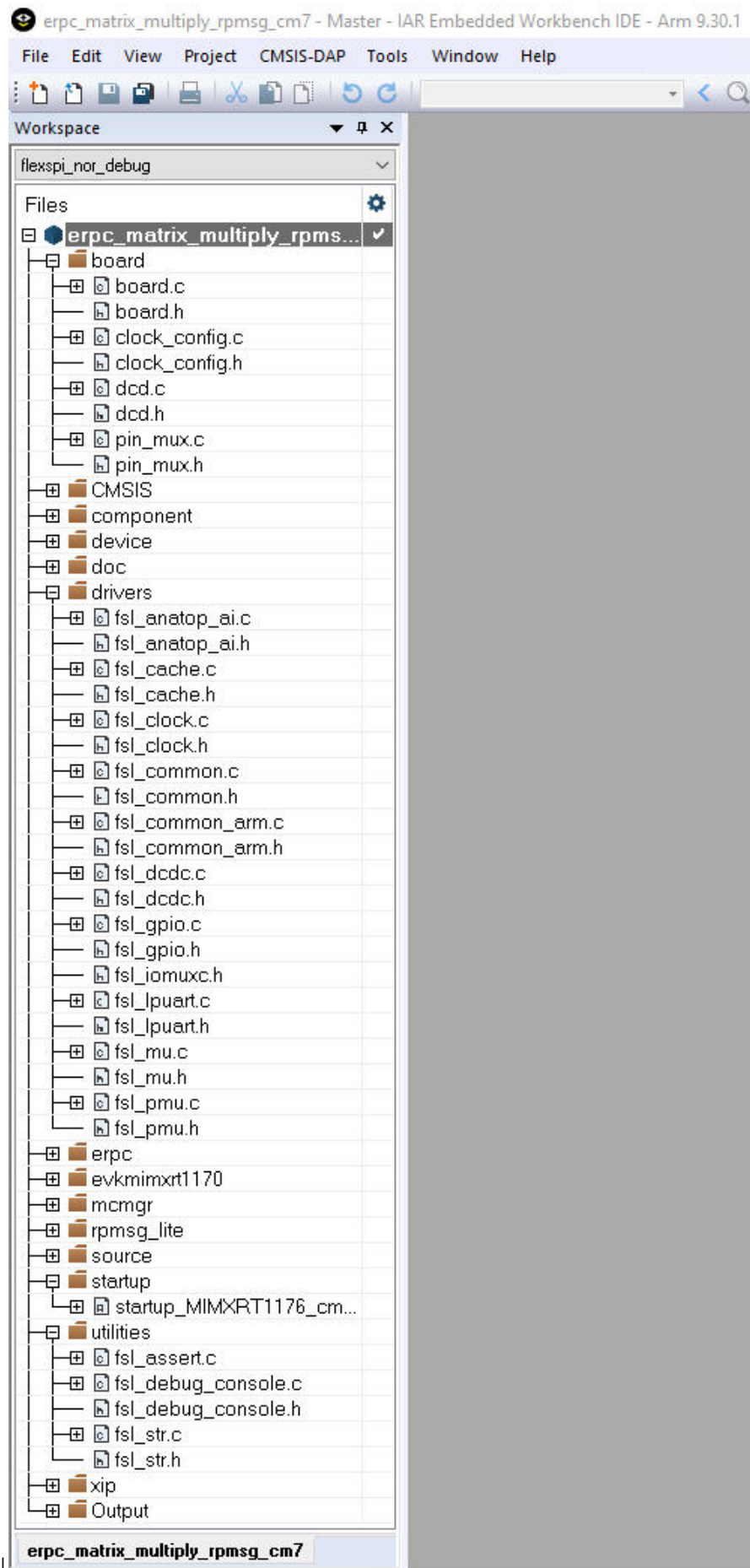
**Multicore client application** The “Matrix multiply” eRPC client project is located in the following folder:

`<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>/`



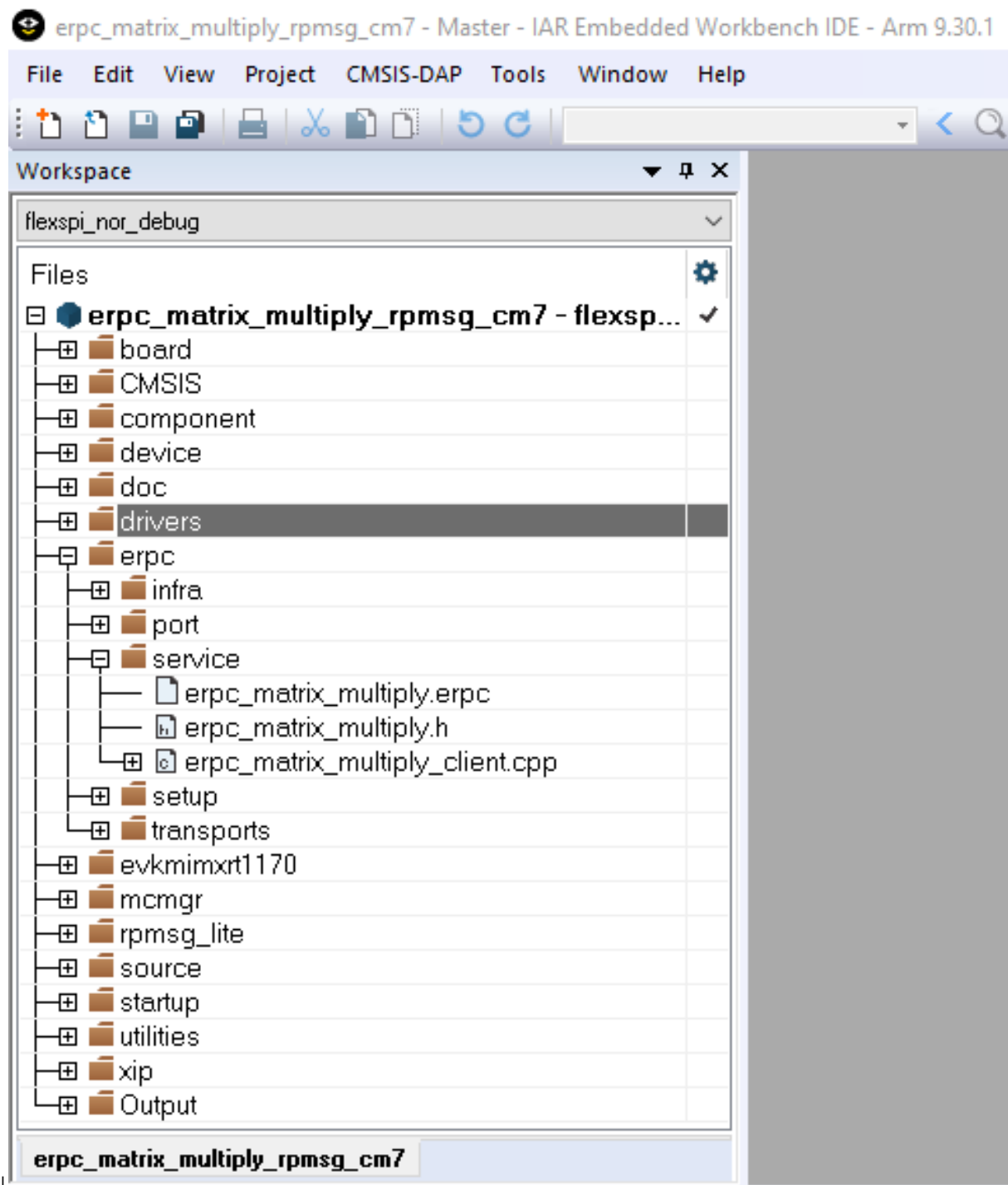
|

**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.

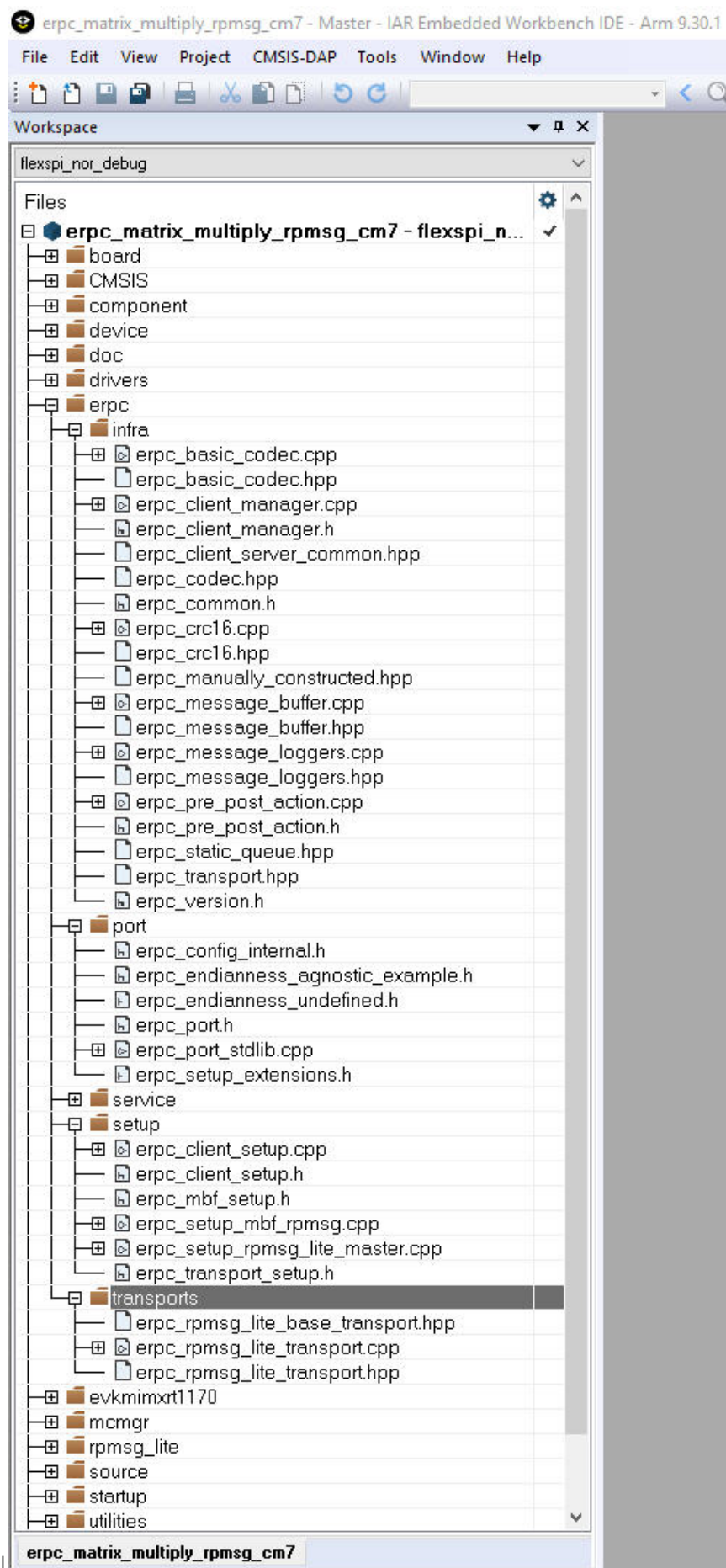
- `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
- `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
- `erpc_config_internal.h` internal eRPC configuration file.

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.



|

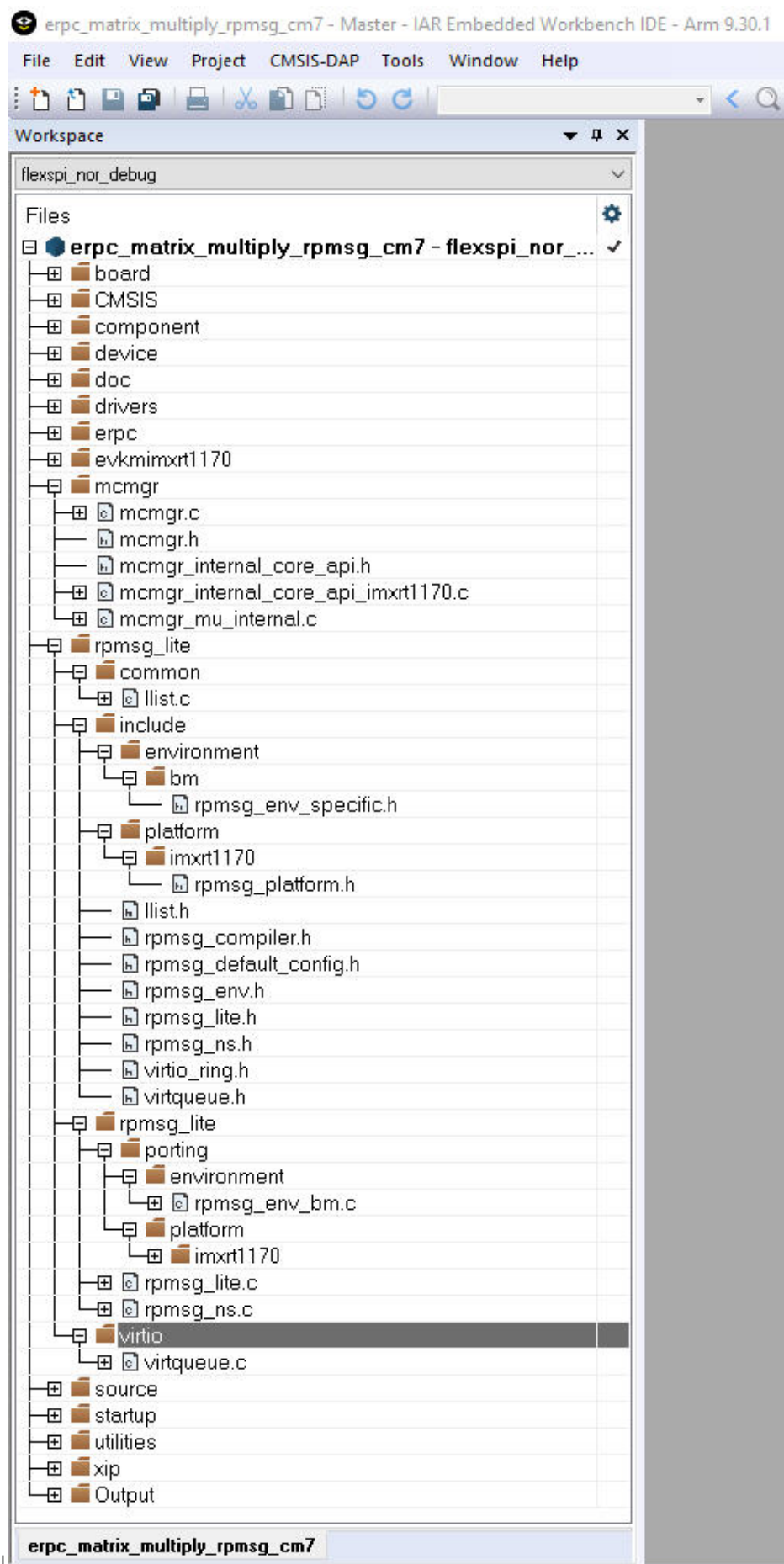
**Parent topic:** Multicore client application

**Client multicore infrastructure files** Because of the RPSMsg-Lite (transport layer), it is also necessary to include RPSMsg-Lite related files, which are in the following folder:

*<MCUXpressoSDK\_install\_dir>/middleware/multicore/rpsmsg\_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/*





**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.

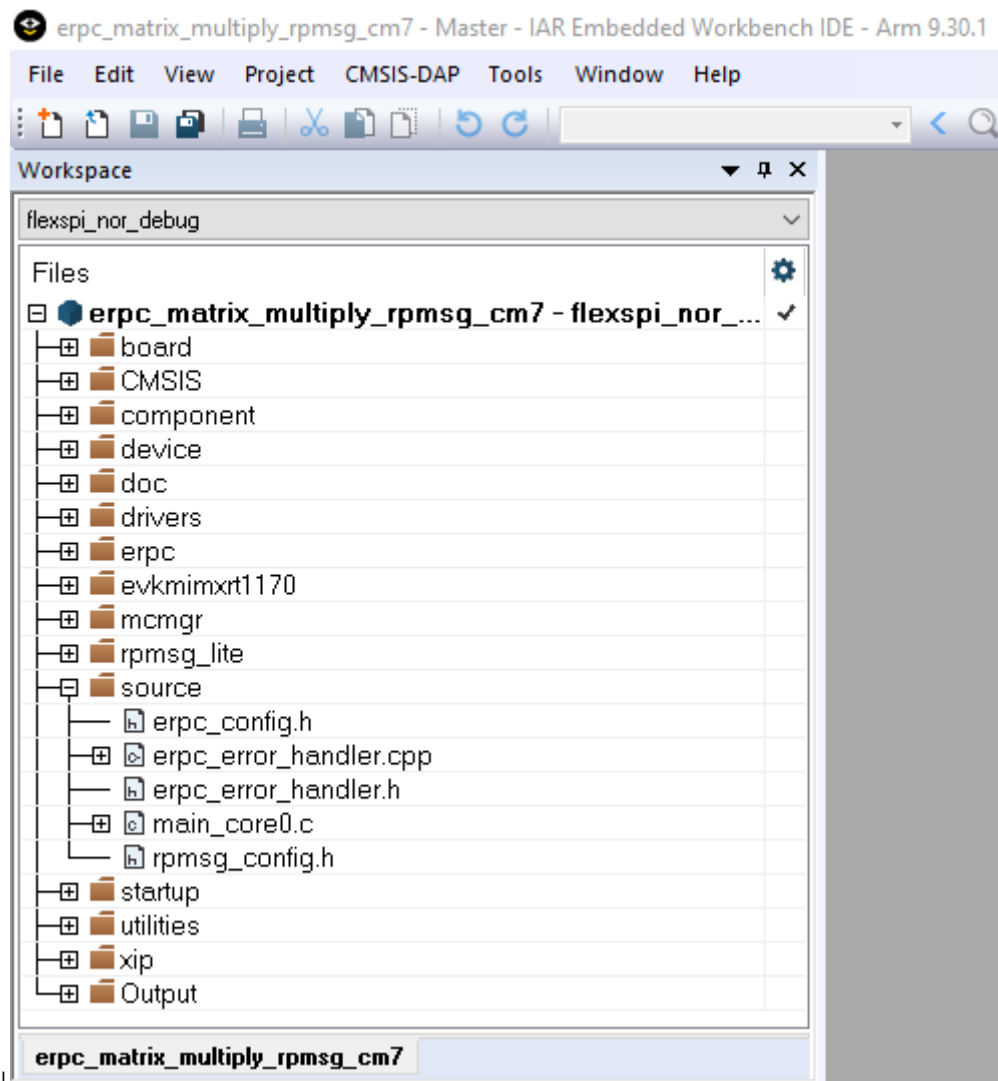
The matrix multiplication can be issued repeatedly, when pressing a software board button.

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 occurred 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 RPMs-Lite). The RPMs-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/transport/erpc\_uart\_cmsis\_transport.hpp

<eRPC base directory>/erpc\_c/transport/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 RPSMsg-Lite). The RPSMsg-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:

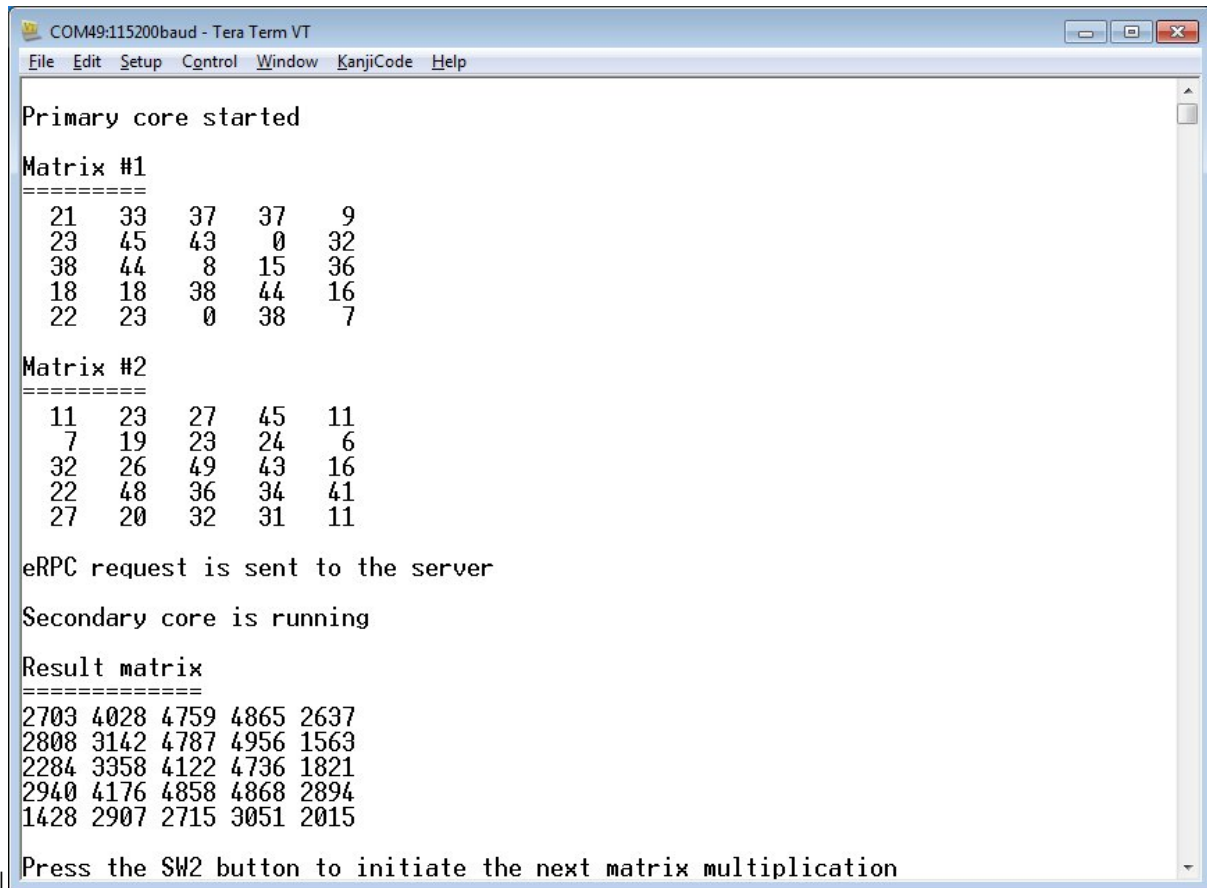
```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver
↳operations */
erpc_transport_t transport;
transport = erpc_transport_cmsis_uart_init((void *)&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 occurred 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:



```

COM49:115200baud - Tera Term VT
File Edit Setup Control Window KanjiCode Help

Primary core started

Matrix #1
=====
 21 33 37 37 9
 23 45 43 0 32
 38 44 8 15 36
 18 18 38 44 16
 22 23 0 38 7

Matrix #2
=====
 11 23 27 45 11
 7 19 23 24 6
 32 26 49 43 16
 22 48 36 34 41
 27 20 32 31 11

eRPC request is sent to the server

Secondary core is running

Result matrix
=====
2703 4028 4759 4865 2637
2808 3142 4787 4956 1563
2284 3358 4122 4736 1821
2940 4176 4858 4868 2894
1428 2907 2715 3051 2015

Press the SW2 button to initiate the next matrix multiplication

```

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](https://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

### 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 Message-Buffer, 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 LIBUSBIO.GetNumPorts API use with libusb 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: MUPort 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 distinguish which file can and cannot be 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 of 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 LIBUSB\_SIO 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 transport 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

## 1.3 Wireless

### 1.3.1 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.                                  | <a href="#">ZigBee Base Device Behavior Specification</a>                |
| Getting Started with the FRDM-MCXW71                  | It provides detailed instructions for installing the MCUXpresso SDK for the FRDM-MCXW71 board.                                                       | <a href="#">Getting Started with FRDM-MCXW71</a>                         |
| 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 (`npxzbota.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 [MCUX-DKK32W061GSUG](#)). 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](#)

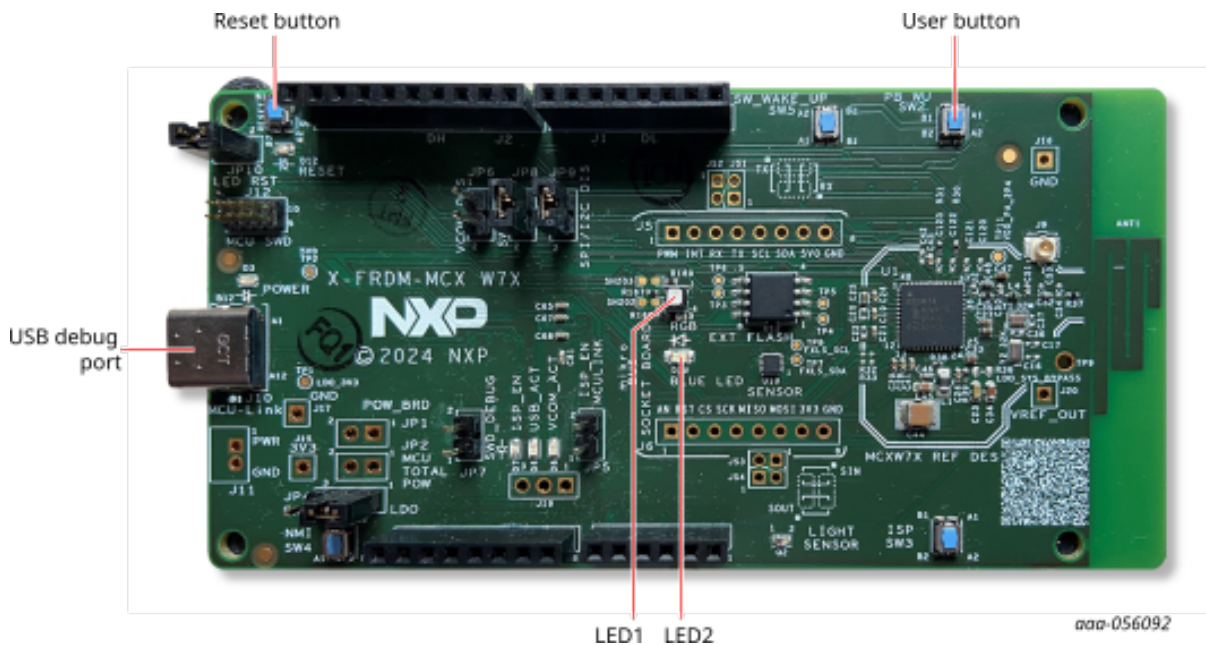
**Examples** The table below lists the example applications provided.

| 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.

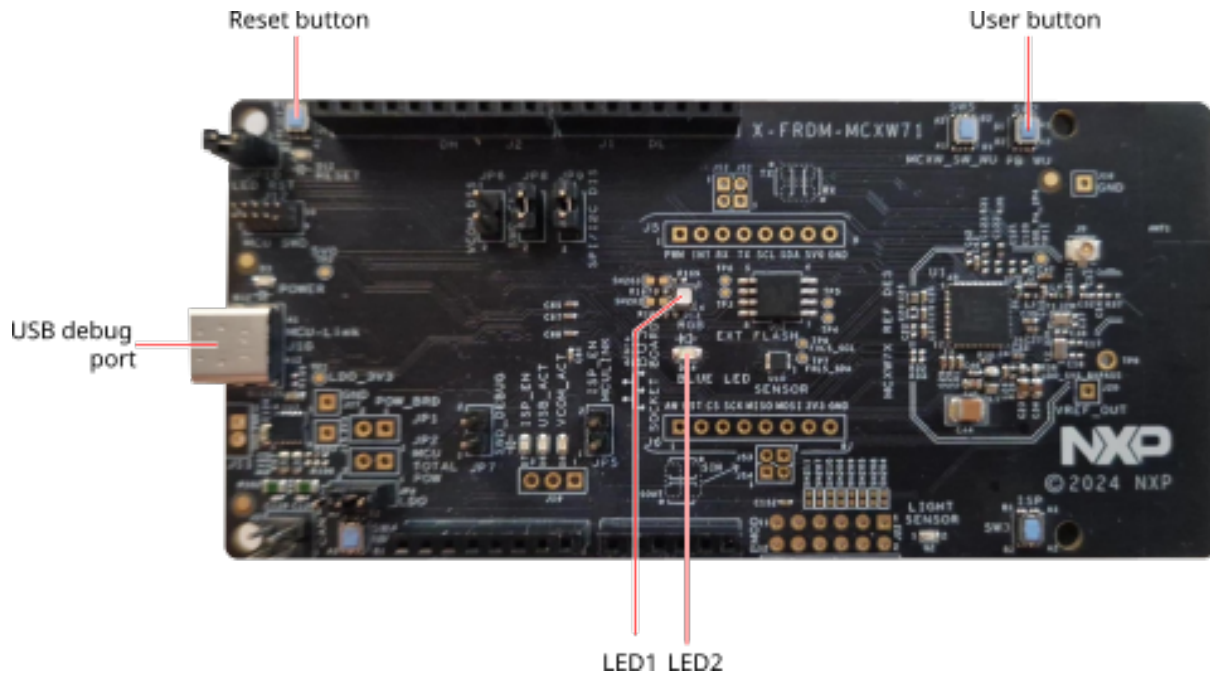
**Note:** The RW612 platform does not support the RX OFF Application use case currently.

**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.



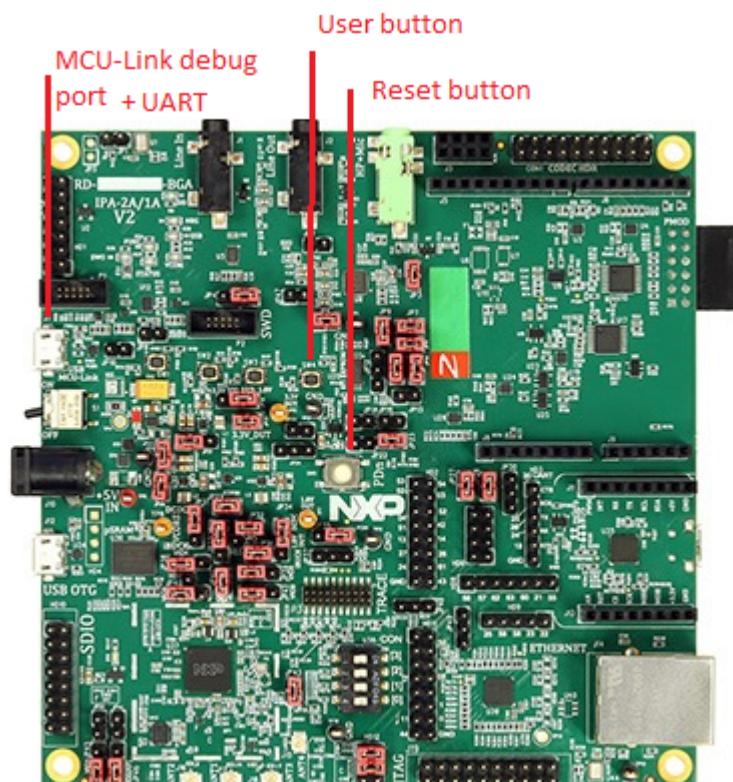
Parent topic: [Examples](#)

**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](#)

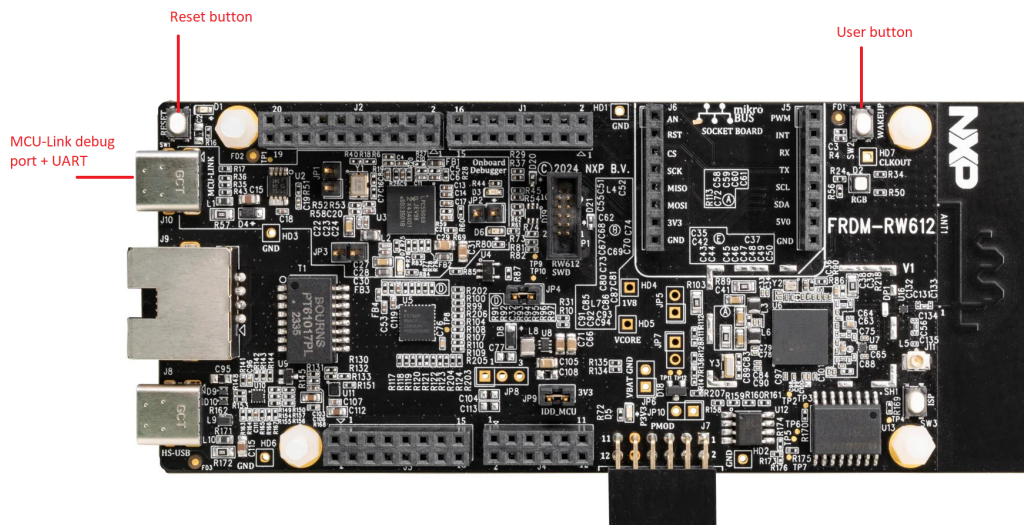
**RD-RW612-BGA board** The FRDM-RW612 board can be used for the ZigBee examples described in this document. The important LEDs and user control buttons are highlighted, as shown in the figure, **RD-RW612-BGA board** below.



Parent topic: [Examples](#)

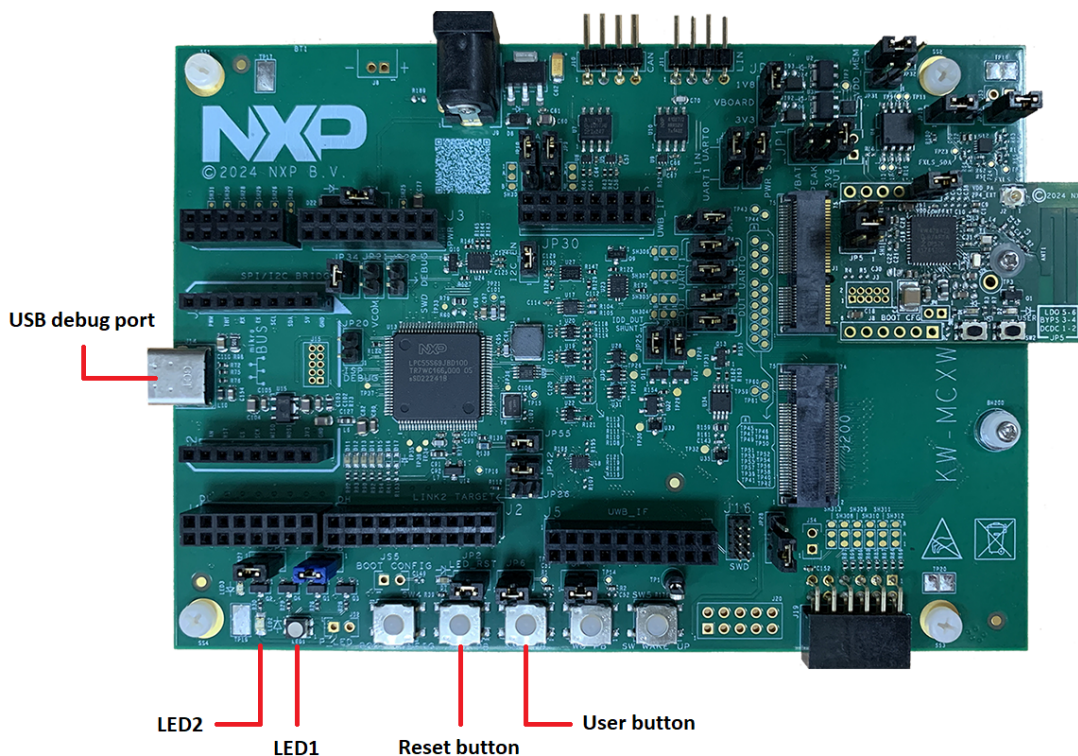


**FRDM-RW612 board** The FRDM-RW612 board is used for the ZigBee examples described in this document. The important LEDs and user control buttons are highlighted, as shown in the figure, **FRDM-RW612 board** below.



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.

| Hard-ware plat-form | Binary files                                                                                                                                                                                                                                                                                                                                                                                                                     |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| FRDM-MCXW board     | <ul style="list-style-type: none"> <li>• <code>frdmmcwx71_zigbee_coordinator_bm.axf</code>• <code>frdmmcwx71_zigbee_ed_rx_on_bm.axf</code>• <code>frdmmcwx71_zigbee_ed_rx_off_bm.axf</code>• <code>frdmmcwx71_zigbee_coordinator_freertos.axf</code>• <code>frdmmcwx71_zigbee_router_freertos.axf</code>• <code>frdmmcwx71_zigbee_ed_rx_on_freertos.axf</code>• <code>frdmmcwx71_zigbee_ed_rx_off_freertos.axf</code></li> </ul> |
| FRDM-MCXW board     | <ul style="list-style-type: none"> <li>• <code>frdmmcwx72_zigbee_coordinator_freertos.axf</code>• <code>frdmmcwx72_zigbee_router_freertos.axf</code>• <code>frdmmcwx72_zigbee_ed_rx_on_freertos.axf</code>• <code>frdmmcwx72_zigbee_ed_rx_off_freertos.axf</code></li> </ul>                                                                                                                                                     |
| MCX-W71-EVK board   | <ul style="list-style-type: none"> <li>• <code>mcxw71evk_zigbee_coordinator_bm.axf</code>• <code>mcxw71evk_zigbee_router_bm.axf</code>• <code>mcxw71evk_zigbee_coordinator_freertos.axf</code>• <code>mcxw71evk_zigbee_router_freertos.axf</code>• <code>mcxw71evk_zigbee_ed_rx_on_freertos.axf</code>• <code>mcxw71evk_zigbee_ed_rx_off_freertos.axf</code></li> </ul>                                                          |
| MCX-W72-EVK board   | <ul style="list-style-type: none"> <li>• <code>mcxw72evk_zigbee_coordinator_freertos.axf</code>• <code>mcxw72evk_zigbee_router_freertos.axf</code>• <code>mcxw72evk_zigbee_ed_rx_on_freertos.axf</code>• <code>mcxw72evk_zigbee_ed_rx_off_freertos.axf</code></li> </ul>                                                                                                                                                         |

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                                                   |
|----------------------------|----------------------------------------------------------|
| <code>toggle</code>        | Sends an On/Off Toggle command to bound devices          |
| <code>steer</code>         | Triggers Network Steering for a device on the network    |
| <code>form</code>          | Triggers network formation for a device not on a network |
| <code>find</code>          | Triggers Finding and Binding as an initiator             |
| <code>factory reset</code> | Factory resets the device, erasing persistent data       |
| <code>soft reset</code>    | 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

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

| Hard-ware plat-form | Binary files                                                                                                                                                                                                                                                                                                                                             |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| FRDM-MCXW7 board    | <ul style="list-style-type: none"> <li>• frdmmcxw71_zigbee_router_bm_v1.axf</li> <li>• frdmmcxw71_zigbee_ed_rx_on_bm_v1.axf</li> <li>• frdmmcxw71_zigbee_ed_rx_off_bm_v1.axf</li> <li>• frdmmcxw71_zigbee_router_freertos_v1.axf</li> <li>• frdmmcxw71_zigbee_ed_rx_on_freertos_v1.axf</li> <li>• frdmmcxw71_zigbee_ed_rx_off_freertos_v1.axf</li> </ul> |
| FRDM-MCXW72 board   | <ul style="list-style-type: none"> <li>• frdmmcxw72_zigbee_router_freertos_v1.axf</li> <li>• frdmmcxw72_zigbee_ed_rx_on_freertos_v1.axf</li> <li>• frdmmcxw72_zigbee_ed_rx_off_freertos_v1.axf</li> </ul>                                                                                                                                                |
| MCX-W71-EVK board   | <ul style="list-style-type: none"> <li>• mcxw71evk_zigbee_router_bm_v1.axf</li> <li>• mcxw71evk_zigbee_ed_rx_on_bm_v1.axf</li> <li>• mcxw71evk_zigbee_ed_rx_off_bm_v1.axf</li> <li>• mcxw71evk_zigbee_router_freertos_v1.axf</li> <li>• mcxw71evk_zigbee_ed_rx_on_freertos_v1.axf</li> <li>• mcxw71evk_zigbee_ed_rx_off_freertos_v1.axf</li> </ul>       |
| MCX-W72-EVK board   | <ul style="list-style-type: none"> <li>• mcxw72evk_zigbee_router_freertos_v1.axf</li> <li>• mcxw72evk_zigbee_ed_rx_on_freertos_v1.axf</li> <li>• mcxw72evk_zigbee_ed_rx_off_freertos_v1.axf</li> </ul>                                                                                                                                                   |

The table below shows that the OTA images are the V2/V3.ota files.

### V2/V3 OTA files

| Hard-ware plat-form                                       | OTA files                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-----------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| FRDM-MCXW71 board                                         | <ul style="list-style-type: none"> <li>• 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</li> </ul> |
| FRDM-MCXW72 board                                         | <ul style="list-style-type: none"> <li>• 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</li> </ul>                                                                                                                                                                                                                                   |
| MCX-W71-EVK board                                         | <ul style="list-style-type: none"> <li>• 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</li> </ul>             |
| MCX-W72-EVK board                                         | <ul style="list-style-type: none"> <li>• 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</li> </ul>                                                                                                                                                                                                                                         |
| Parent topic: <a href="#">Zigbee Over-The-Air Upgrade</a> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

**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 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 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 *italic* type.

Code fragments are represented in the Courier New typeface.

**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)             |
| APSD    | Application Support (sub-layer) Data        |
| APSM    | 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.

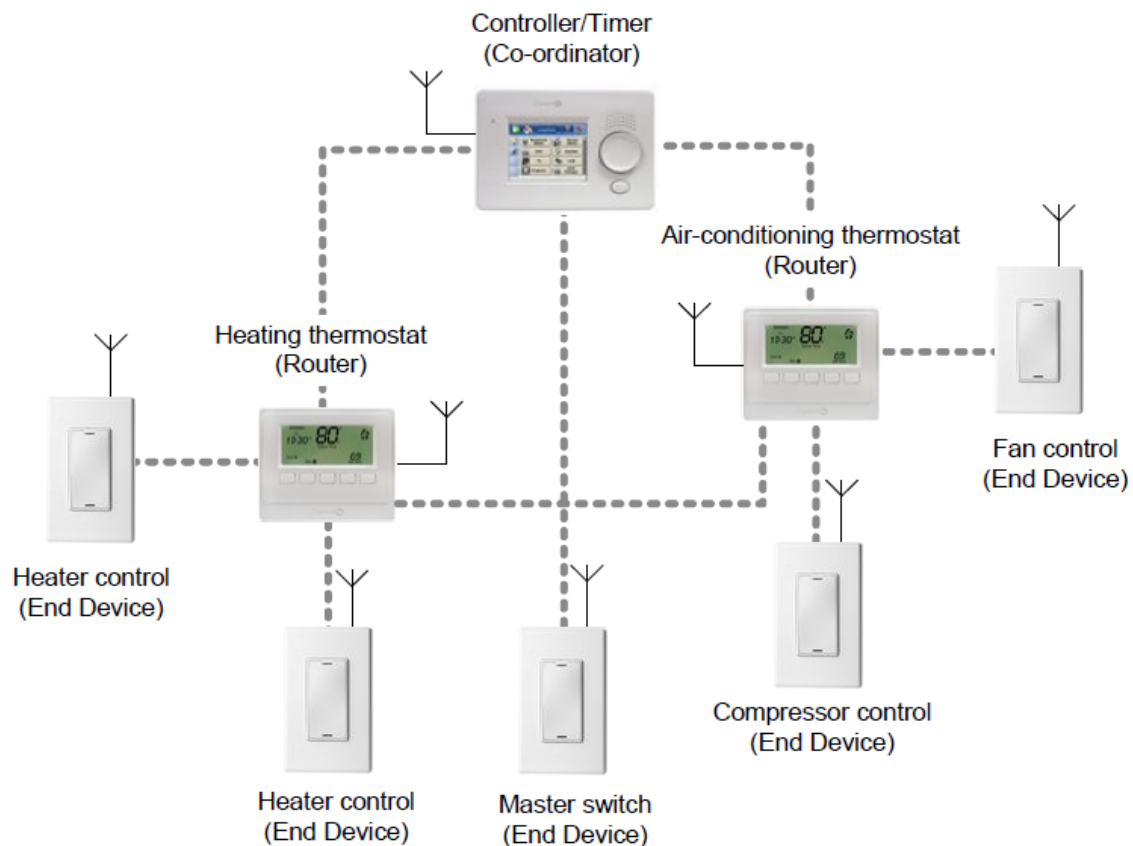
Figure 1 shows a simple example of a ZigBee network in an HVAC (Heating, Ventilation, and Air-Conditioning) system.

**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](#)

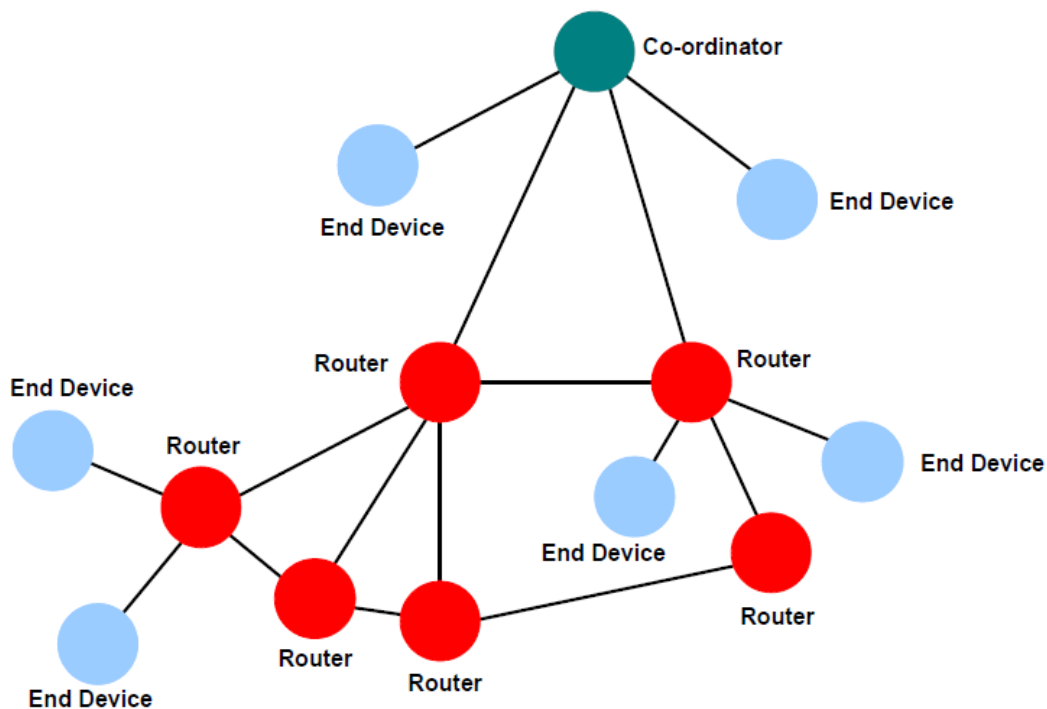
**ZigBee PRO network topology** ZigBee facilitates a range of network topologies from the simplest Star topology, through the highly structured Tree topology to the flexible Mesh topology. ZigBee PRO is designed primarily for Mesh networks.

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                 |
| <b>Total</b>      | 90                 |

#### Channel distribution across pages

| Channel Page | Description |
|--------------|-------------|
| 863-876 MHz  | 63          |
| 915-921 MHz  | 27          |
| <b>Total</b> | 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](#)

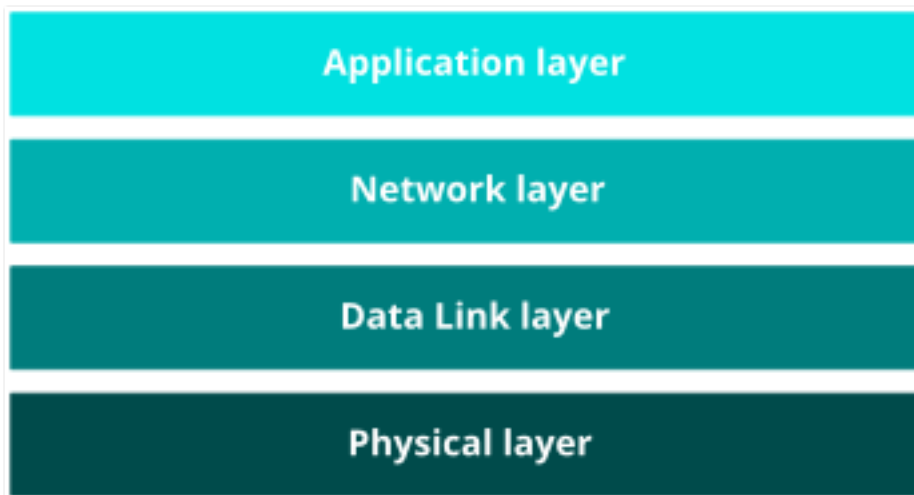
**ZigBee PRO architecture and operation** This chapter introduces ZigBee PRO from architectural and operational view-points by describing:

- 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 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

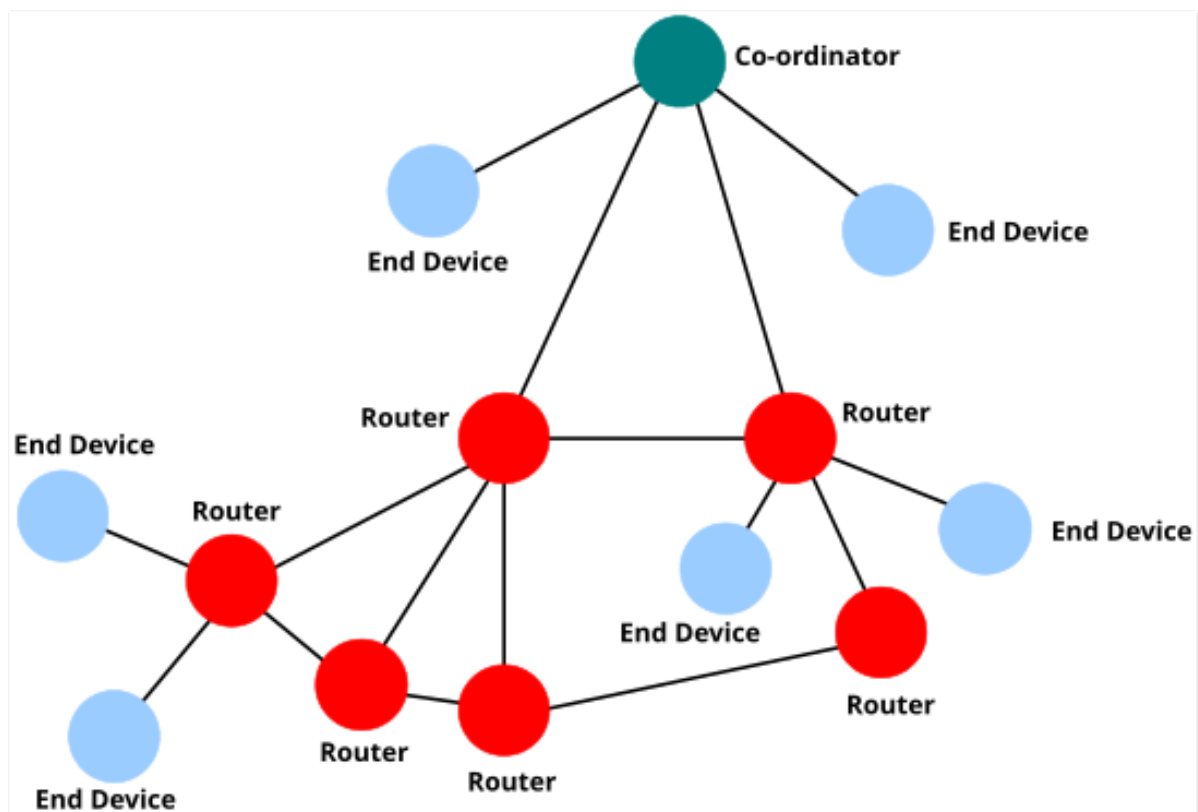
**Network topology** The ZigBee PRO standard was designed to facilitate wireless networks with the Mesh topology.

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 Coordinator 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 itself). 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.

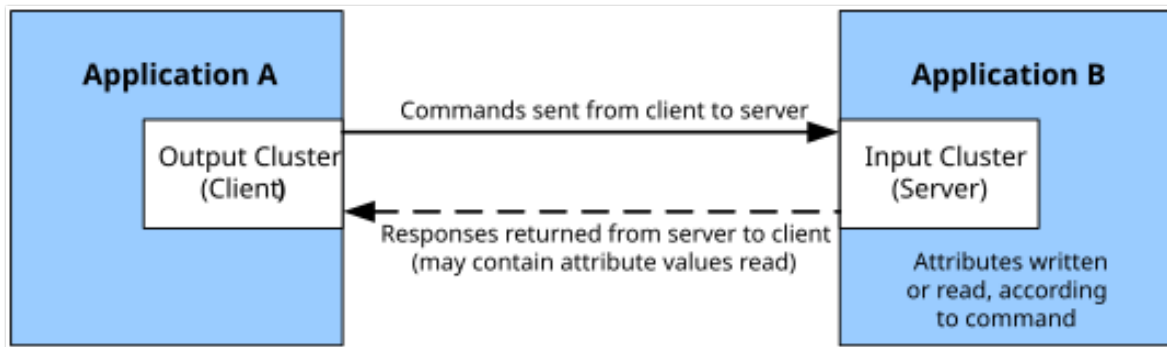
A cluster corresponds to a specific piece of functionality for a device application. The total functionality for the application is determined by the ZigBee device type that it implements and the clusters that the device type uses (see Section 3.4.4). Thus, clusters are the functional building blocks 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 *'Mt-gRoomLight'*.

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

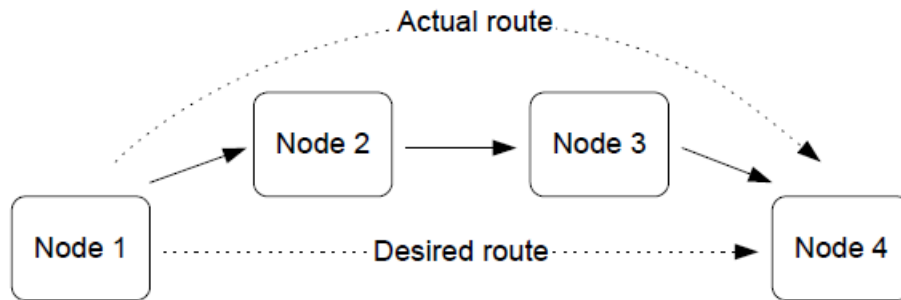
**Parent topic:**[ZigBee PRO architecture and operation](#)

**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.

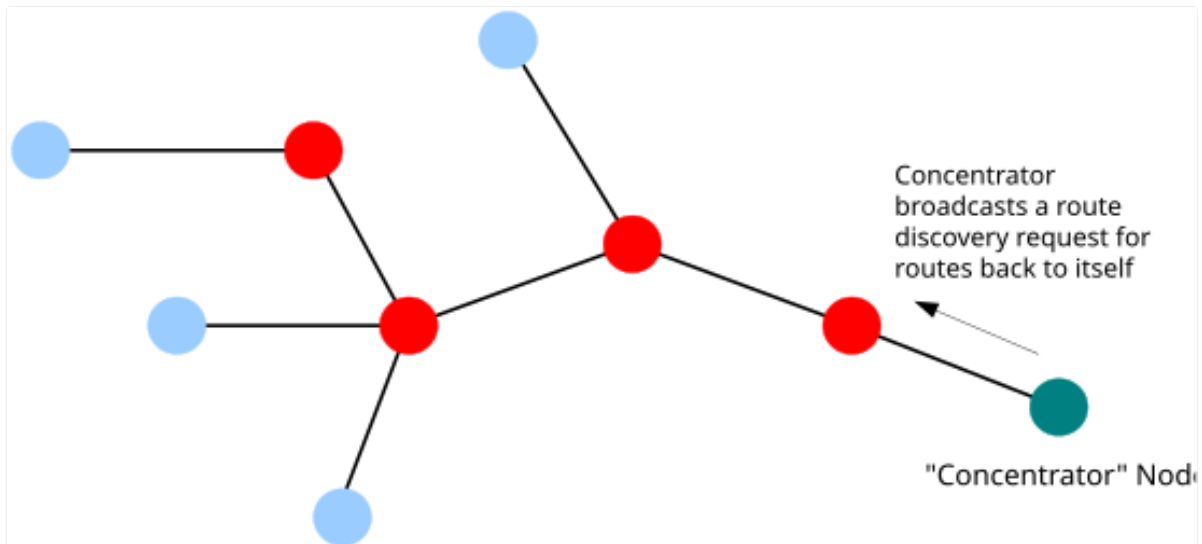
In order to establish communication with the concentrator, each remote node may initiate a ‘route discovery’, resulting in a corresponding entry in the Routing table of each routing node along the way. If most network nodes need to communicate with the concentrator, many such route discoveries may be 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.

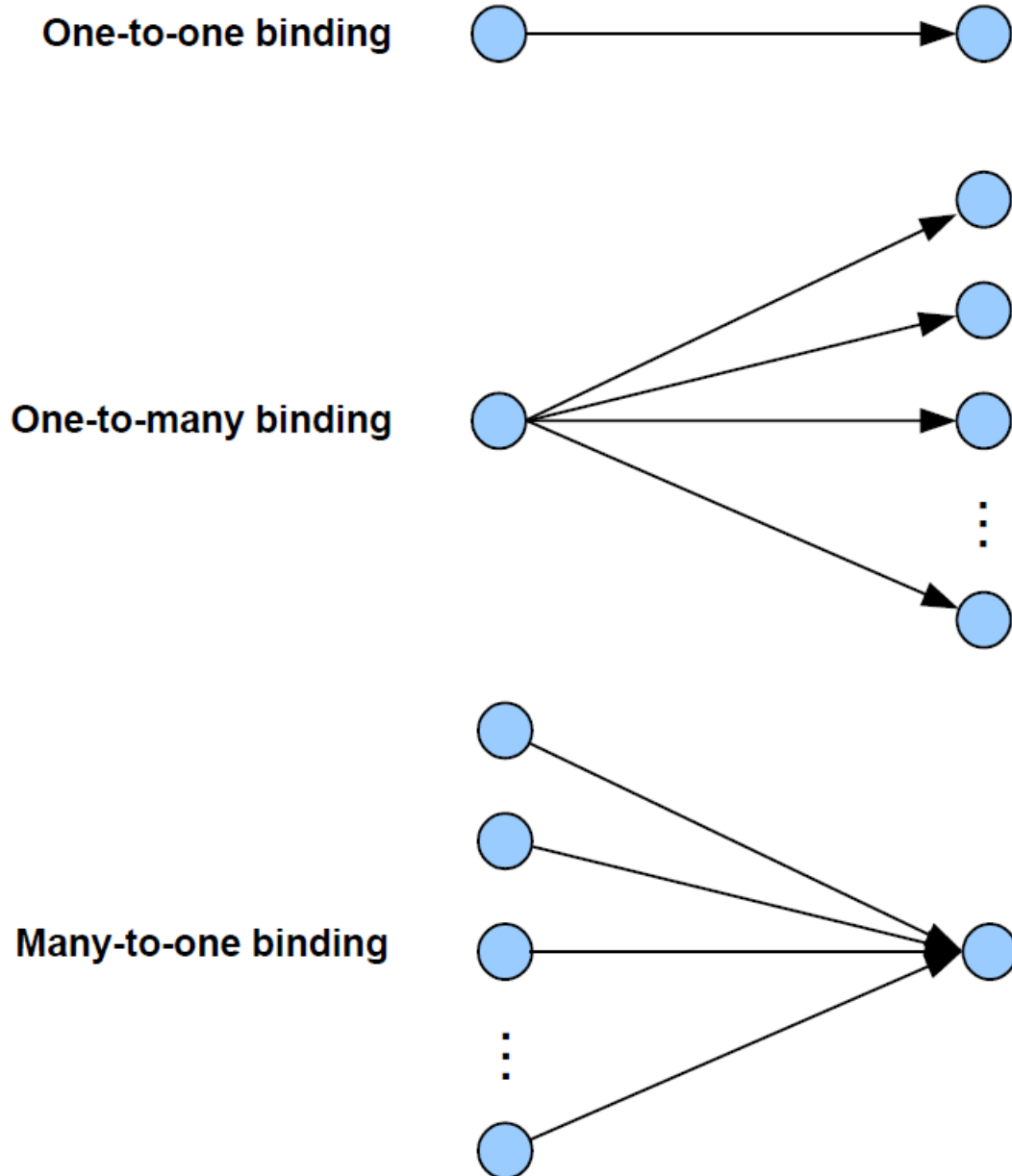
The binding between two applications is specified by:

- 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 Section 2.6), 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.

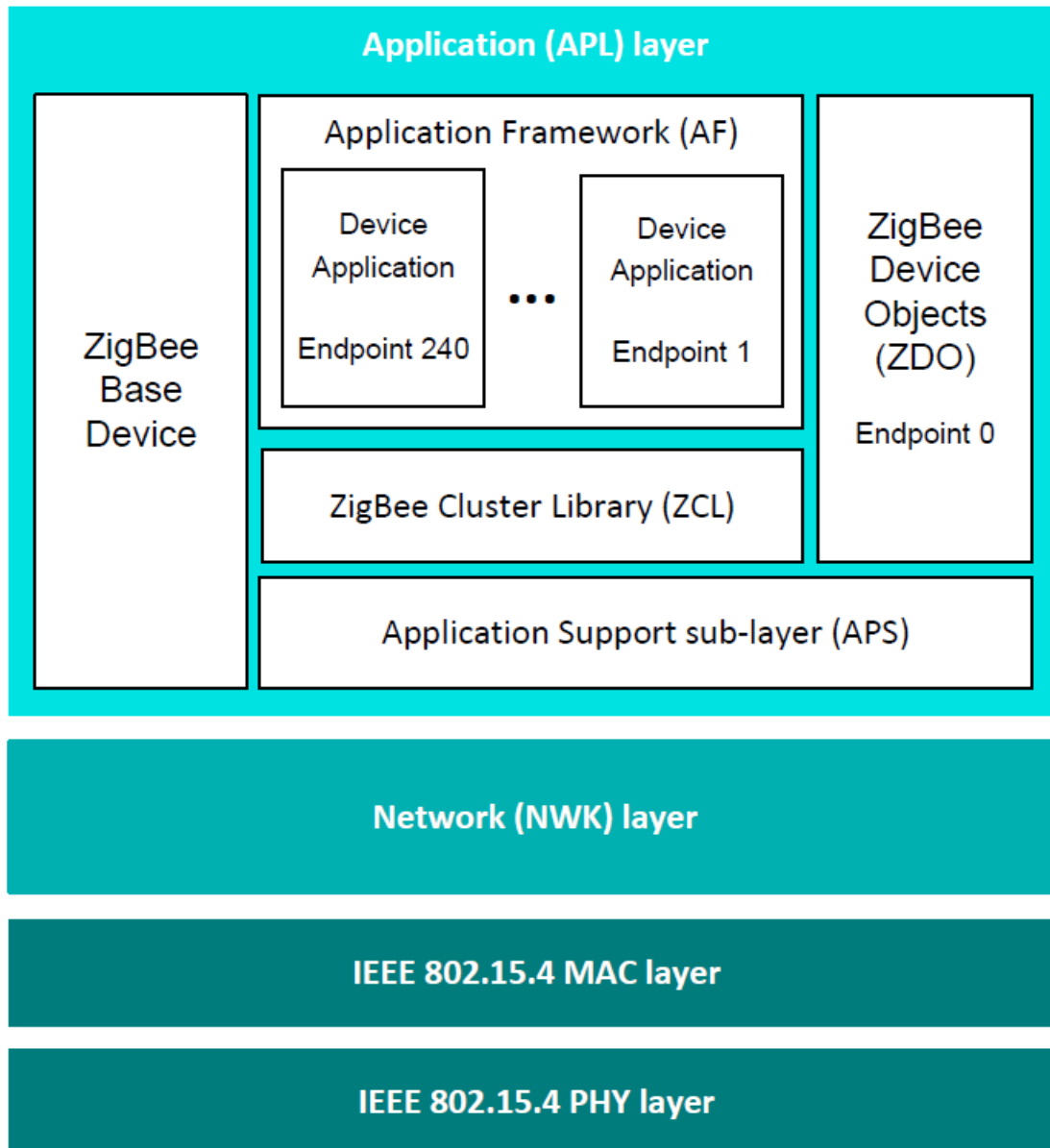
- Custom system: Bindings are created manually by the system integrator or installation technician, who may use a graphical software tool to draw binding lines between clusters on nodes.

**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

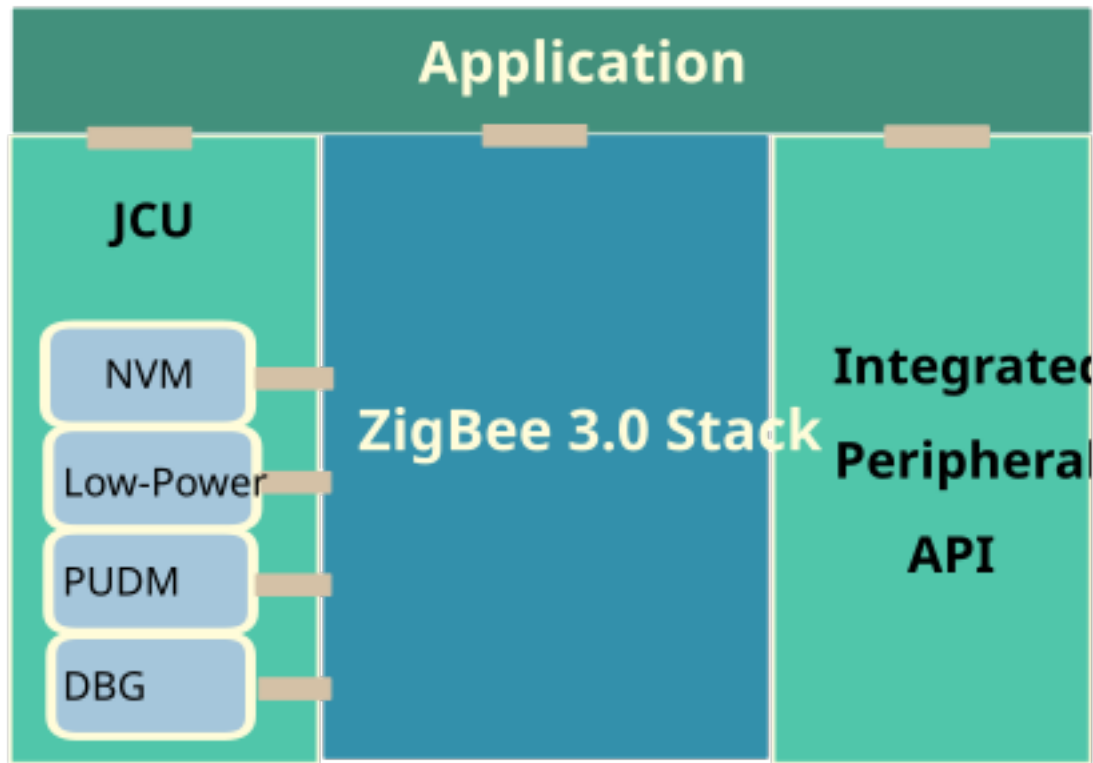
**Parent topic:**Detailed architecture

**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).

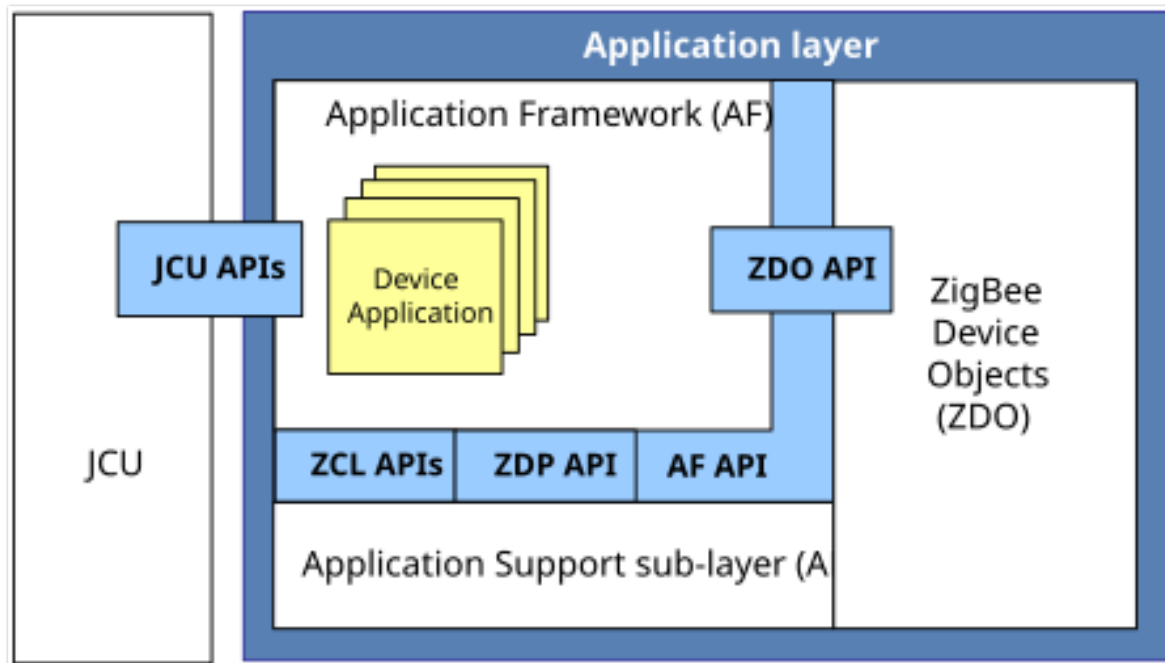
**ZigBee PRO APIs** The ZigBee PRO APIs are concerned with network-specific operations and easy interaction with the ZigBee PRO stack from the application code. These C-function APIs are supplied in the ZigBee 3.0 SDK (see Section 5.1, Development environment and resources).

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                           | <b>ZDO API:</b> Network formation and local network management<br><b>ZDP API:</b> Network discovery and remote network management | -                                                           |
| Basic data transfer                                                                           | <b>AF API:</b> Sending and receiving data messages                                                                                | <b>PDUM API:</b> Assembling and disassembling data messages |
| Binding endpoints for data transfers between them                                             | <b>ZDO API:</b> Basic binding<br><b>ZDP API:</b> Manipulation of remote Binding tables                                            | -                                                           |
| Low-power modes (Sleep and Doze)                                                              | -                                                                                                                                 | <b>Low-Power API:</b> Managing low-power modes              |
| Preserving context data (for example, for resuming operation after sleep without memory held) | -                                                                                                                                 | <b>NVM API:</b> Saving and restoring context data           |
| Network security                                                                              | <b>ZDO API:</b> 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

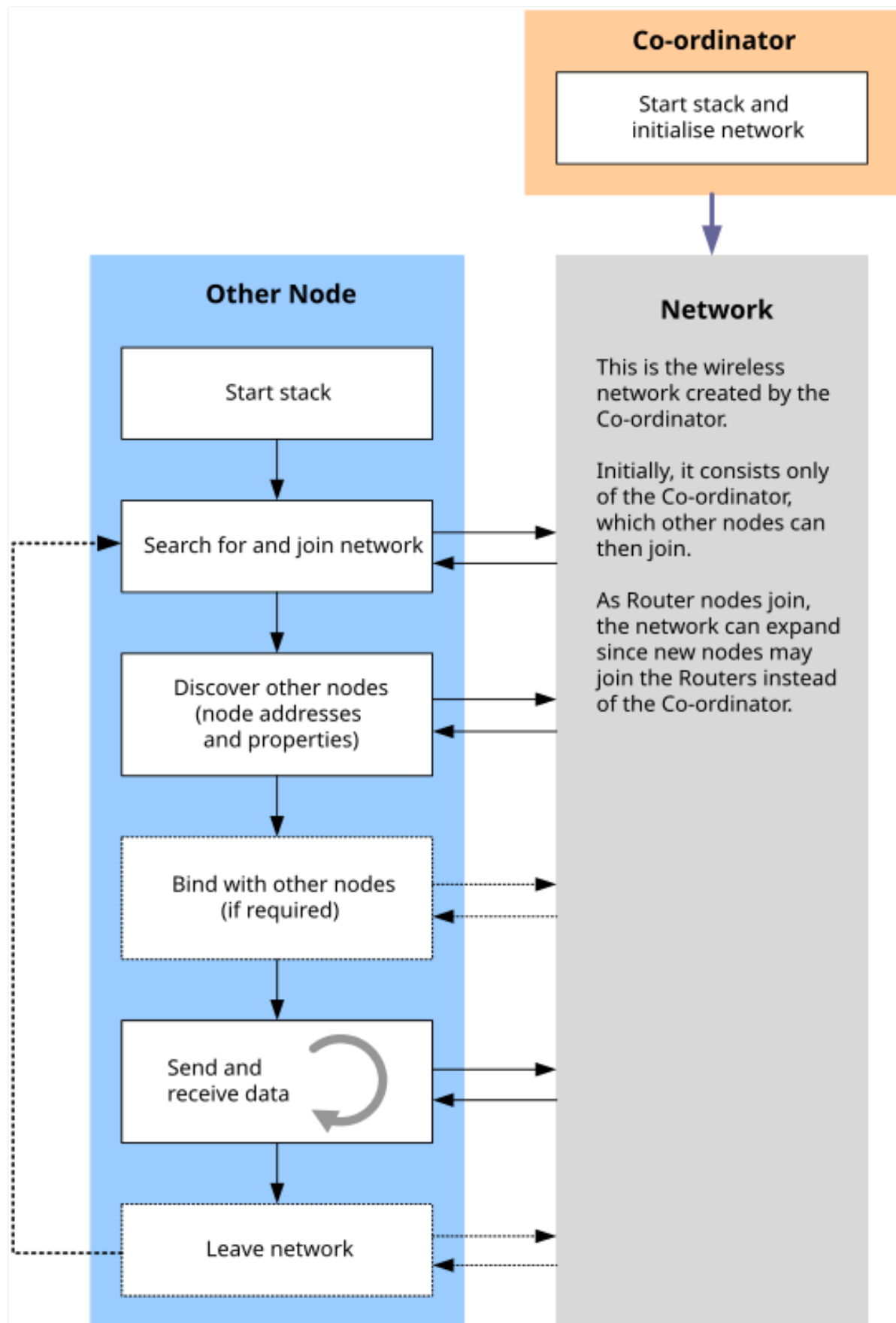
node that initiated the request. Once received, this response message can be collected using the function **ZQ\_bZQueueReceive()** - see General queue management.

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**



**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 (encryption/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 be 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 Section 11.2.5) 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**

| Se-<br>cu-<br>rity<br>Key                 | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Netwo<br>level<br>Se-<br>cu-<br>rity      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Network<br>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<br>level<br>Se-<br>cu-<br>rity     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Global<br>key<br>(pre-<br>config)         | - 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<br>link<br>key                     | Optional key used to encrypt communications between a pair of nodes - may be one of the below categories:                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Pre-<br>config<br>unique<br>link<br>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<br>Centre<br>Link<br>Key<br>(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                                                                                                                                                                                                            |
| Ap-<br>pli-<br>ca-<br>tion<br>link<br>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\_eAplAfnit()** 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](#).

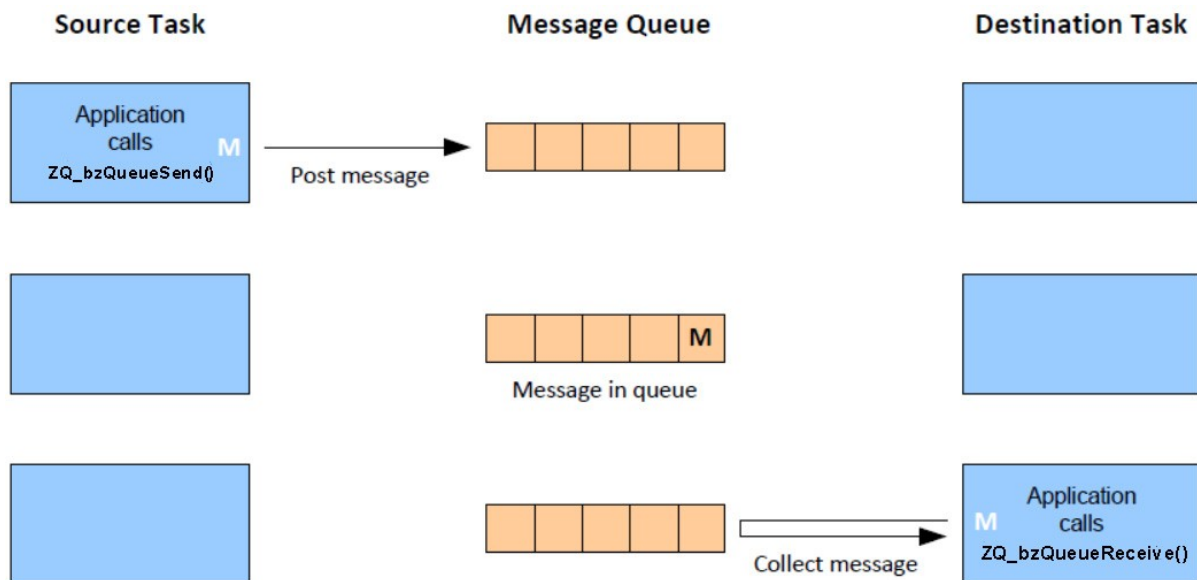
**Message queues** Communications between application tasks on a node are implemented via message queues. The application can create a dedicated message queue for a particular communication channel. A set of functions are provided to implement message queues, as indicated in Section 6.9.1.1 below (these functions are detailed in Section 10.1). 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 Section 101.2.1).

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:

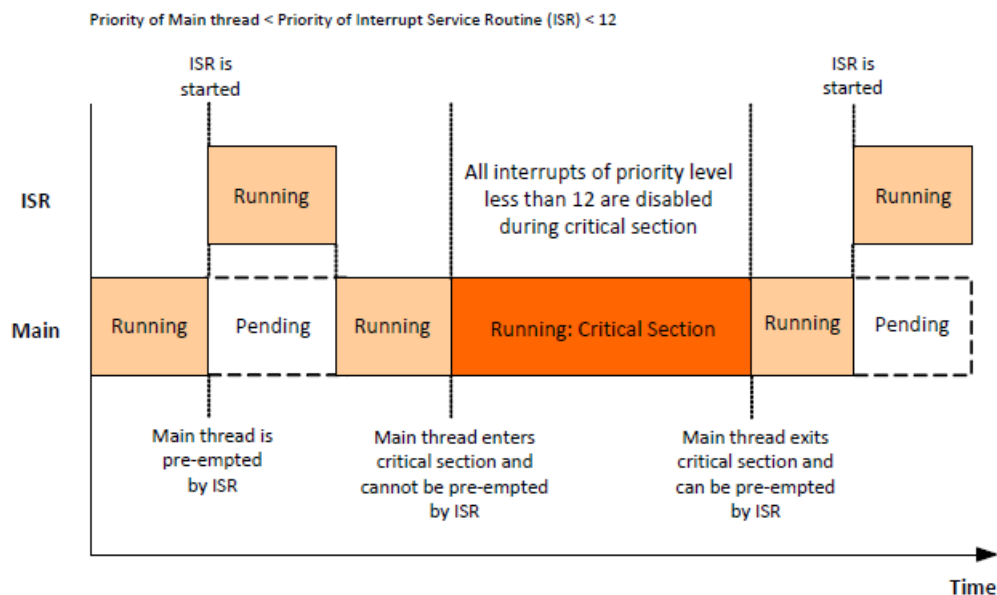
- **Critical Section:** A section of application code can be designated as a ‘critical section’, which means that the execution of this code section cannot be preempted by an interrupt with a priority level less than 12. A critical section should be short in order to avoid suspending interrupts for a 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 `u8Level` before 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 `u8Level` before 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_eAplAffinit()`.

**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\_eAplAfnit()**, 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                                                                        |
|-----------------------------------------|------------------------------------------------------------------------------------|
| <b>zps_DEVICE_PERMISSIONS_ALL_PERM</b>  | Allow all requests from other nodes                                                |
| <b>zps_DEVICE_PERMISSIONS_JOIN_DISA</b> | Do not allow join requests from other nodes                                        |
| <b>zps_DEVICE_PERMISSIONS_DATA_RE</b>   | Do not allow data requests from other nodes and disable end-to-end acknowledgments |

When a device joins the network, the **ALL\_PERMITTED** 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_PERMITTED` 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
- *psActvNtEntryPointer* 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\_PERMITTED)
- 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\_PERMITTED
- 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\_PERMITTED)
- 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\_PERMITTED
  - 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_tsAfFlashInfoSet` structure.
- Information about an individual link key is stored in RAM in a read-only `ZPS_TclkDescriptorEntry` structure, 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 `u16ExtAddrLkup` from 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 Section 3.2.4). 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 *pu64Address* and 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 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\_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\_teZdoNwkKeyState contains 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_KEY                 | No network key should be used.                                                                                                                                                                                                                                    |
| ZPS_ZDO_PRECONFIGURED_LINK_KEY | A pre-configured link key should be used. This key can be fixed at the time of manufacture.                                                                                                                                                                       |
| ZPS_ZDO_DISTANT_LINK_KEY       | 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_PRECONFIGURED_LINK_KEY for Home Automation (HA) compatibility and a ZPS_ZDO_ZLL_LINK_KEY for ZLL networks. |
| ZPS_ZDO_INSTALL_CODE           | 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\_teZdoDeviceType contains 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_PERMITTED = 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_PERM | 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** *pfExtendedStatusCallBackPointer* 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 tsBeaconFilterType structure (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 *psStartParams,
 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

- *psStartParams* 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 *psStartParams);
```

**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** *psStartParams* 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** *psStartParams* 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\_tsAfProfileDataReq** structure (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
- **u16DestAddr:** 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 (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\_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 (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\_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 (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\_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 (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\_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 (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\_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 (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\_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 (encryption/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\_tsAplAfNodeDescriptor is 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:

- eLogicalType contains 3 bits (bits 0-2) indicating the ZigBee device type of the node, as follows:
  - 000: Coordinator
  - 001: Router
  - 010: End Device
- bComplexDescAvail is set to 1 if there is a Complex descriptor available for node.
- bUserDescAvail is set to 1 if there is a User descriptor available for node.
- eReserved is reserved.
- eFrequencyBand contains 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
- eApsFlags is not currently supported and set to zero.
- 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 the table: Table.
- 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 the table: Table.
- `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 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_tsAplAfNodePowerDescriptor` is shown below.

```
typedef struct {
 uint32 eCurrentPowerMode : 4;
 uint32 eAvailablePowerSources : 4;
 uint32 eCurrentPowerSource : 4;
 uint32 eCurrentPowerSourceLevel : 4;
} ZPS_tsAplAfNodePowerDescriptor;
```

where:

- `eCurrentPowerMode` contains 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
- `eAvailablePowerSources` contains 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
- `eCurrentPowerSource` contains 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).
- `eCurrentPowerSourceLevel` contains 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\_tsAplAfSimpleDescriptor is shown below.

```
typedef struct {
 uint16 u16ApplicationProfileId;
 uint16 u16DeviceId;
 uint8 u8DeviceVersion;
 uint8 u8Endpoint;
 uint8 u8InClusterCount;
 uint8 u8OutClusterCount;
 uint16 *pu16InClusterList;
 uint16 *pu16OutClusterList;
} ZPS_tsAplAfSimpleDescriptor;
```

where:

- **u16ApplicationProfileId** is 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).
- **u16DeviceId** is 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.
- **u8DeviceVersion** contains 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).
- **u8Endpoint** is the number, in the range 1-240, of the endpoint to which the Simple descriptor corresponds.
- **u8InClusterCount** is an 8-bit count of the number of input clusters, supported on the endpoint, that will appear in the list pointed to by the **pu16InClusterList** element.
- **u8OutClusterCount** is an 8-bit count of the number of output clusters, supported on the endpoint, that will appear in the **pu16OutClusterList** element.
- **\*pu16InClusterList** is 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.
- **\*pu16OutClusterList** is 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\_tsAfEvent structure is detailed below.

```
typedef struct {
 ZPS_teAfEventType eType;
 ZPS_tuAfEventData uEvent;
} ZPS_tsAfEvent;
```

where

- eType indicates the event type, using the enumerations listed and described in Section 11.1.
- uEvent is 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\_tuAfEventData structure 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\_tsAfDataIndEvent structure 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:

- u8DstAddrMode indicates the type of destination address specified through the element uDstAddress (see the Table below.)
- uDstAddress is the address of the destination node for the data packet (the type of address is specified using the element u8DstAddrMode above).
- u8DstEndpoint is the number of the destination endpoint (in range 0-240).
- u8SrcAddrMode indicates 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.

- `u8SrcAddress` is the address of the source node for the data packet (the type of address is specified using the element `u8SrcAddrMode` above).
- `u8SrcEndpoint` is the number of the source endpoint (in range 1-240).
- `u16ProfileId` is the identifier of the ZigBee device profile of the device which can interpret the data.
- `u16ClusterId` is the identifier of the cluster (which belongs to the device profile specified in `u16ProfileId`) which is capable of interpreting the data.
- `hAPduInst` is the handle of the APDU which contains the data.
- `eStatus` is one of the status codes from the NWK layer or MAC layer, detailed in Section 11.2.3 and Section 11.2.4.
- `eSecurityStatus` indicates 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).
- `u8LinkQuality` is a measure of the signal strength of the radio link over which the data packet was sent (for the last hop).
- `u32RxTime` is 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\_tsAfDataConfEvent structure is detailed below.

```
typedef struct {
 uint8 u8Status;
 uint8 u8SrcEndpoint;
 uint8 u8DstEndpoint;
 uint8 u8DstAddrMode;
 ZPS_tuAddress uDstAddr;
 uint8 u8SequenceNum;
} ZPS_tsAfDataConfEvent;
```

where:

- `u8Status` is one of the status codes from the lower stack layers, detailed in Section 11.2.
- `u8SrcEndpoint` is the number of the (local) source endpoint for the data transfer (in range 1-240).
- `u8DstEndpoint` is the number of the destination endpoint for the data transfer (in range 1-240).
- `u8DstAddrMode` indicates 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.

- `u8DstAddr` is the address of the destination node for the data packet (the type of address is specified using the element `u8DstAddrMode` above).
- `u8SequenceNum` is 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:

- `u8Status` is one of the status codes from the lower stack layers, detailed in Section 11.2.
- `u8SrcEndpoint` is the number of the (local) source endpoint for the data transfer (in range 1-240).
- `u8DstEndpoint` is the number of the destination endpoint for the data transfer (in range 1-240).
- `u8DstAddrMode` indicates 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.
- `u16DstAddr` is 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 `u8DstAddrMode` above).
- `u8SequenceNum` is the sequence number of the request that initiated the data transfer.
- `u16ProfileId` is the identifier of the ZigBee device profile of the device for which the data transfer was intended.
- `u16ClusterId` is 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_tsAfNwkFormationEvent` structure 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\_tsAfNwkJoinedEvent structure 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:

- u16Addr is the 16-bit network address allocated to the joining node.
- bRejoin indicates 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\_tsAfNwkJoinFailedEvent structure is detailed below.

```
typedef struct
{
 uint8 u8Status;
 bool_t bRejoin;
} ZPS_tsAfNwkJoinFailedEvent;
```

where:

- u8Status is one of the status codes from the lower stack layers, detailed in Section 11.2.
- bRejoin indicates 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\_tsAfNwkDiscoveryEvent structure is detailed below.

```
typedef struct
{
 uint32 u32UnscannedChannels;
 uint8 eStatus;
 uint8 u8NetworkCount;
 uint8 u8SelectedNetwork;
 ZPS_tsNwkNetworkDescr *psNwkDescriptors;
} ZPS_tsAfNwkDiscoveryEvent;
```

where:

- u32UnscannedChannels is 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.

- `estatus` is 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.
- `u8NetworkCount` is the number of networks that had been discovered when this event was generated.
- `u8SelectedNetwork` is the index of the recommended network in the array of reported networks (see below).
- `psNwkDescriptors` is 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_tsAfNwkJoinIndEvent` structure 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:

- `u64ExtAddr` is the 64-bit IEEE (MAC) address of the joining node.
- `u16NwkAddr` is the 16-bit network address assigned to the joining node.
- `u8Capability` is a bitmap indicating the operational capabilities of the joining node. This bitmap is detailed in Table below.
- `u8Rejoin` indicates the method used to join the network:
  - 0x00 if joined through association.
  - 0x01 if joined directly or used orphaning.
  - 0x02 if was network rejoin.
- `u8SecureRejoin` indicates 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:

- u64ExtAddr is 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
- u8Rejoin indicates 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:

- u64ExtAddr is the 64-bit IEEE (MAC) address of the leaving node. This value is zero if the local node itself is leaving.
- eStatus is 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\_tsAfNwkStatusIndEvent structure is detailed below.

```
typedef struct {
 uint16 u16NwkAddr;
 uint8 u8Status;
} ZPS_tsAfNwkStatusIndEvent;
```

where:

- u16NwkAddr is the 16-bit network address of the node associated with the event.
- u8Status is 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\_tsAfNwkRouteDiscoveryConfEvent structure is detailed below.

```
typedef struct {
 uint16 u16DstAddress;
 uint8 u8Status;
 uint8 u8NwkStatus;
} ZPS_tsAfNwkRouteDiscoveryConfEvent;
```

where:

- u16DstAddress is the destination address for which the route discovery confirm event was generated.
- u8Status is one of the status codes from the MAC layer, detailed in Section 11.2.4.
- u8NwkStatus is 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\_tsAfPollConfEvent structure is detailed below.

```
typedef struct {
 uint8 u8Status;
} ZPS_tsAfPollConfEvent;
```

where u8Status is 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\_tsAfNwkEdScanConfEvent structure is defined as:

```
typedef ZPS_tsNwkNlmeCfmEdScan ZPS_tsAfNwkEdScanConfEvent;
```

where ZPS\_tsNwkNlmeCfmEdScan is 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\_tsAfErrorEvent structure 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;
 } sAfErrorApu;
 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\_tsAfErrorEvent structure are listed and described below.

#### eError Enumerations

| eError<br>tion    | Enumera-<br>tion | Description                                                                                                                                             |
|-------------------|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| ZPS_ERROR_APDU_TC |                  | Allocated APDU instance is too small to accommodate received message. This error is detailed in the structure sAfErrorApu, 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 sAfErrorApu, 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 sAfErrorApu, 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.    |

#### sAfErrorApu

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:

- `u16ProfileId` is the identifier of the ZigBee application profile associated with the source and destination endpoints for the message.
- `u16ClusterId` is the identifier of the cluster associated with the source and destination endpoints for the message.
- `u16SrcAddr` is the 16-bit network address of the source node of the message.
- `u16DataSize` is the size of the received message, in bytes.
- `hAPdu` is the handle of the local APDU pool from which the APDU instance comes.
- `u8SrcEndpoint` is the number of the source endpoint of the message.
- `u8DstEndpoint` is 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 `hMessage` is 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

- `uDstAddr` is the address of the remote node for the binding (the type of address is specified using the element `u8DstAddrMode` above).
- `u8DstAddrMode` indicates 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_tsAfZdoUnbindEvent` structure is defined as:

```
typedef ZPS_tsAfZdoBindEvent ZPS_tsAfZdoUnbindEvent;
```

where `ZPS_tsAfZdoBindEvent` is 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_tsAfZdoLinkKeyEvent` structure is defined as:

```
typedef struct {
 uint64 u64IeeeLinkAddr;
} ZPS_tsAfZdoLinkKeyEvent;
```

where `u64IeeeLinkAddr` is 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_tsAfBindRequestServerEvent` structure is detailed below.

```
typedef struct {
 uint8 u8Status;
 uint8 u8SrcEndpoint;
 uint32 u32FailureCount;
} ZPS_tsAfBindRequestServerEvent;
```

where:

- `u8Status` is 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 `u32FailureCount` below).
- `u8SrcEndpoint` is the number of the local endpoint from which the data packet was sent.
- `u32FailureCount` is 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\_tsAfInterPanDataIndEvent structure 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

- sDstAddr is a structure of the type ZPS\_tsInterPanAddress (see Section 8.2.3.3) which contains the PAN ID and address for the destination node(s) of the inter-PAN data packet.
- u8SrcAddrMode indicates the type of address specified through the element u64SrcAddress (see Table).
- u16SrcPan is the PAN ID of the network from which the data packet originates.
- u64SrcAddress is the address of the node which sent the data packet (the type of address is specified using the element u8SrcAddrMode above).
- u16ProfileId is the identifier of the application profile for which the data packet is intended.
- u16ClusterId is the identifier of the cluster for which the data packet is intended.
- hAPduInst is the handle of the APDU instance for the data packet.
- eStatus is one of the status codes from the lower stack layers, detailed in Section 11.2.
- u8DstEndpoint is the number of the destination endpoint for the data packet (in range 1-240).
- u8LinkQuality is 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\_tsAfInterPanDataConfEvent structure is detailed below.

```
typedef struct {
 uint8 u8Status;
 uint8 u8Handle;
} ZPS_tsAfInterPanDataConfEvent;
```

where

- u8Status is one of the status codes from the lower stack layers, detailed in Section 11.2.
- u8Handle is 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:

- u8SequNumber is the sequence number of the ZDP request/response
- u16ClusterId is the ID of the cluster to which the request/response relates
- uZdpData is a union of the different ZDP request/response types:
  - sDeviceAnnce is a structure of the type ZPS\_tsAplZdpDeviceAnnceReq, described in Section 9.2.2.3
- sMgmtNwkUpdateReq is a structure of the type ZPS\_tsAplZdpMgmtNwkUpdateReq, described in Section 9.2.2.41
- sPermitJoiningReq is a structure of the type ZPS\_tsAplZdpMgmtPermitJoiningReq, described in Section 9.2.3.39
- sDiscoveryCacheRsp is 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 Section 9.2.2.27
- sRemoveBkupBindEntryRspis a structure of the type ZPS\_tsAplZdpRemoveBkupBindEntryRsp, described in Section 9.2.2.28
- 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 Section 9.2.3.13
- 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_tsNwkNetworkDescr` structure 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:

- `u64ExtPanId` is the Extended PAN ID of the discovered network.
- `u8LogicalChan` is the 2400-MHz channel on which the network was found.
- `u8StackProfile` is 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
- `u8ZigBeeVersion` is the ZigBee version of the discovered network.
- `u8PermitJoining` indicates the number of detected nodes with ‘permit joining’ enabled (and therefore allowing nodes to join the network through them).
- `u8RouterCapacity` indicates the number of detected nodes that are allowing Routers to join the network through them.
- `u8EndDeviceCapacity` indicates 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_tsNwkNlmeCfmEdScan` structure is detailed below.

```
typedef struct
{
 uint8 u8Status;
 uint8 u8ResultListSize;
 uint8 au8EnergyDetect[ZPS_NWK_MAX_ED_RESULTS];
} ZPS_tsNwkNlmeCfmEdScan;
```

where

- `u8Status` is one of the status codes from the lower stack layers, detailed in Section 11.2.
- `u8ResultListSize` is 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_tsInterPanAddress` structure 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:

- `eMode` is used to specify the type of destination address that will be used in the field `uAddress` below. 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
- `u16PanId` is the PAN ID of the destination network - a value `0xFFFF` can be used to specify a broadcast to all reachable ZigBee PRO networks
- `uAddress` is the address of the target node(s) in the destination network (the address type must be as specified above in the `eMode` field) - 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_tsAfProfileDataReq` structure 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:

- `uDstAddr` is the address of the destination node for the transmission request (can be 16- or 64-bit, as specified by `eDstAddrMode`).
- `u16ClusterId` is the Cluster ID of the destination cluster.
- `u16ProfileId` is the Profile ID of the destination application profile.
- `u8SrcEp` is the source endpoint number (1-240) on the local node.
- `eDstAddrMode` is 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).
- `u8DstEp` is the destination endpoint number (1-240) on the remote node.
- `eSecurityMode` is 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).
- `u8Radius` is 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 `tsBeaconFilterType` structure is detailed below.

```
typedef struct
{
 uint64 *pu64ExtendPanIdList;
 uint16 u16Panid;
 uint16 u16FilterMap;
 uint8 u8ListSize;
 uint8 u8Lqi;
 uint8 u8Depth;
} tsBeaconFilterType;
```

where:

- `pu64ExtendPanIdList` is 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 `u8FilterMap` bitmap:
  - 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
- `u16Panid` is a 16-bit PAN ID on which beacons can be filtered
- `u8ListSize` is the number of Extended PAN IDs in the list pointed to by

pu64ExtendPanIdList

- u8Lqi is 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 u8FilterMap bitmap
- u8Depth is the tree depth of the neighbor device. A value of 0x00 indicates that the device is the ZigBee coordinator for the network.
- u16FilterMap is an 16-bit bitmap detailing the filtering requirements, as follows:

| Bit | Enumera-<br>tion  | Description                                                                                                                                                                                                                                |
|-----|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0   | BF_BITMAP_        | If set, field pu64ExtendPanIdList points to a blacklist of networks.                                                                                                                                                                       |
| 1   | BF_BITMAP_ (0x2)  | If set, field pu64ExtendPanIdList points 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 u16Panid can 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:

- u32OutgoingFrameCounter is the outgoing frame counter value which is incremented on each transmission to a destination address below.
- u16ExtAddrLkup is 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:

- `u64ExtPanId` is the Extended PAN ID of the distributed security network.
- `pu8NwkKey` is a pointer to a location to receive the network key.
- `u16PanId` is the PAN ID of the network.
- `u16NwkAddr` is the network address of the local node.
- `u8KeyIndex` is the sequence number required to identify the network key in the security set.
- `u8LogicalChannel` is the number of the radio channel 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:**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:

- `u16SectorSize` is the size, in bytes, of the Flash memory sector used to store the link keys.
- `u16CredNodesCount` is the maximum number of nodes for which link keys can be stored in the Flash memory sector.
- `u8SectorSet` is 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_t u16CredOffset;
 uint16_t u16TclkRetries;
} ZPS_TclkDescriptorEntry;
```

where:

- **u16CredOffset** is the offset, in bytes, of the storage location for the node's link key in the relevant Flash memory sector.
- **u16TclkRetries** is 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 re-joined 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 Section 9.2.3.9).

**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 Section 9.2.3.11).

**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(\n
 PDUM_thAPduInstance hAPduInst,\n
 uint8 *pu8SeqNumber,\n
 ZPS_tsAplZdpFindNodeCacheReq\n
 *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 {\n
 uint16 u16NwkAddr;\n
 uint64 u64IeeeAddr;\n
} 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 Section 9.2.3.28).

**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 Section 8.2.2.36).

```
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 Section 8.2.3.36).

**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 Section 8.2.3.39).

```
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 Section 9.2.2.40).

```
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 Section 9.2.2.41).

```
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 <code>u32ScanChannels</code> . The time, in seconds, spent scanning each channel is determined by the value of <code>u8ScanDuration</code> and the number of scans is equal to the value of <code>u8ScanCount</code> . Valid for unicasts only. |
| 0x06-0xFF | Reserved                                                                                                                                                                                                                                                                                                            |
| 0xFE      | Change radio channel to single channel specified through <code>u32ScanChannels</code> and set the network manager address to that specified through <code>u16NwkManagerAddr</code> . Valid for broadcasts only.                                                                                                     |
| 0xFF      | Update the stored radio channel mask with that specified through <code>u32ScanChannels</code> (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\_tsAfZdpEvent structure 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 Section 9.2.1
- 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\_tsAplZdpNodeDescBitFields structure 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:

- eFrequencyBand is 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
- eApsFlags is a 3-bit value containing flags that indicate the ZigBee APS capabilities of the node (not currently supported and should be set to 0).
- eReserved is a 3-bit reserved value.
- bUserDescAvail is a 1-bit value indicating whether a User descriptor is available for the node - 1 indicates available, 0 indicates unavailable.
- bComplexDescAvail is a 1-bit value indicating whether a Complex descriptor is available for the node - 1 indicates available, 0 indicates unavailable.
- eLogicalType is 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\_tsAplZdpNodePowerDescriptor is shown below.

```
typedef struct {
 union
 {
 ZPS_tsAplZdpPowerDescBitFields sBitFields;
 uint16 u16Value;
 } uBitUnion;
} ZPS_tsAplZdpNodePowerDescriptor;
```



where:

- `sBitFields` is a structure of type `ZPS_tsAplZdpPowerDescBitFields` (described below) containing various items of information about the node's power.
- `u16value` is used for the union and should be set to `0x0000`.

**ZPS\_tsAplZdpPowerDescBitFields** The `ZPS_tsAplZdpPowerDescBitFields` structure 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:

- `eCurrentPowerSourceLevel` is 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)
- `eCurrentPowerSource` is 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
- `eAvailablePowerSource` is 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).
- `eCurrentPowerMode` is 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_tsAplZdpSimpleDescType` is 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:

- **u8Endpoint** is the number, in the range 1-240, of the endpoint to which the Simple descriptor corresponds.
- **u16ApplicationProfileId** is 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).
- **u16DeviceId** is 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.
- **sBitFields** is a structure of type **ZPS\_tsAplZdpSimpleDescBitFields** (described below) containing information about the endpoint.
- **u8Value** is used for the union and must be set to 0x00.
- **u8InClusterCount** is an 8-bit count of the number of input clusters, supported on the endpoint, that will appear in the list pointed to by the **pu16InClusterList** element.
- **\*pu16InClusterList** is 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.
- **u8OutClusterCount** is an 8-bit count of the number of output clusters, supported on the endpoint, that will appear in the **pu16OutClusterList** element.
- **\*pu16OutClusterList** is 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\_tsAplZdpSimpleDescBitFields** structure 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:

- **eDeviceVersion** is a 4-bit value identifying the version of the device description supported by the endpoint.
- **eReserved** is 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\_tsAplZdpNwkAddrReq structure 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\_tsAplZdpIEEEAddrReq structure is detailed below.

```
typedef struct {
 uint16 u16NwkAddrOfInterest; uint8 u8RequestType;
```

```
uint8 u8StartIndex;
} ZPS_tsAplZdpIEEEAddrReq;
```

where:

- u16NwkAddrOfInterest is the network address of the node of interest
- u8RequestType is 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
- 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\_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\_tsAplZdpDeviceAnnceReq structure is detailed below.

```
typedef struct { uint16 u16NwkAddr; uint64 u64IeeeAddr;
uint8 u8Capability;
} ZPS_tsAplZdpDeviceAnnceReq;
```

where:

- u16NwkAddr is the network address of the sending node
- u64IeeeAddr is the IEEE address of the sending node
- u8Capability is 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\_tsAplZdpNodeDescReq structure is detailed below.

```
typedef struct {
uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpNodeDescReq;
```

where u16NwkAddrOfInterest is 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\_tsAplZdpPowerDescReq structure is detailed below.

```
typedef struct {
 uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpPowerDescReq;
```

where `u16NwkAddrOfInterest` is 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_tsAplZdpSimpleDescReq` structure is detailed below.

```
typedef struct {
 uint16 u16NwkAddrOfInterest;
 uint8 u8EndPoint;
} ZPS_tsAplZdpSimpleDescReq;
```

where:

- `u16NwkAddrOfInterest` is the network address of the node of interest.
- `u8EndPoint` is 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_tsAplZdpExtendedSimpleDescReq` structure is detailed below.

```
typedef struct { uint16 u16NwkAddr; uint8 u8EndPoint;
 uint8 u8StartIndex;
} ZPS_tsAplZdpExtendedSimpleDescReq;
```

where:

- `u16NwkAddrOfInterest` is the network address of the node of interest
- `u8EndPoint` is the number of the relevant endpoint on the node (1-240)
- `u8StartIndex` is 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_tsAplZdpComplexDescReq` structure is detailed below.

```
typedef struct {
 uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpComplexDescReq;
```

where `u16NwkAddrOfInterest` is 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:

- u16NwkAddr is the network address of the node of interest
- u8StartIndex is 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:

- u16NwkAddrOfInterest is the network address of the node of interest
- u8Length is the length of the User descriptor
- szUserDescriptor is 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 u16ServerMask is 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:

- u16NwkAddr is the network address of the sending node
- u64IeeeAddr is 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:

- u16NwkAddr is the network address of the sending node
- u64IeeeAddr is the IEEE address of the sending node
- u8NodeDescSize is the size of the Node descriptor to store
- u8PowerDescSize is the size of the Power descriptor to store
- u8ActiveEpSize is the size of the list of active endpoints to store
- u8SimpleDescCount is the number of Simple descriptors to store
- pu8SimpleDescSizeList is 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:

- u16NwkAddr is the network address of the sending node
- u64IeeeAddr is the IEEE address of the sending node
- sNodeDescriptor is a pointer to the Node descriptor to store (this is itself a structure of the type ZPS\_tsAplZdpNodeDescriptor, detailed in Section 8.2.1.1)

**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:

- u16NwkAddr is the network address of the sending node
- u64IeeeAddr is the IEEE address of the sending node
- sPowerDescriptor is a pointer to the Power descriptor to store (this is itself a structure of the type ZPS\_tsAplZdpNodePowerDescriptor, detailed in Section 8.2.1.2)

**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:

- u16NwkAddr is the network address of the sending node
- u64IeeeAddr is the IEEE address of the sending node
- u8Length is the length of the Simple descriptor to store
- sSimpleDescriptor is a pointer to the Simple descriptor to store (this is itself a structure of the type ZPS\_tsAplZdpSimpleDescType, detailed in Section 8.2.1.3)

**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:

- u16NwkAddr is the network address of the sending node
- u64IeeeAddr is the IEEE address of the sending node
- u8ActiveEPCount is the number of active endpoints in the list to store
- pu8ActiveEpList is 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:

- u16NwkAddr is the network address of the node of interest
- u64IeeeAddr is 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:

- u16NwkAddr is the network address of the End Device of interest
- u64IeeeAddr is 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:

- **u16BindingTarget** is the network address of the node to hold the binding (either a node with primary binding table cache or the local node).
- **u64SrcIeeeAddress** is the IEEE address of the local node.
- **u8SrcEndpoint** is the number of the local endpoint to be bound (1-240).
- **u16ProfileId** is the application profile ID to be matched for the binding.
- **u8NumInClusters** is the number of input clusters of the local endpoint (available for matching with output clusters of remote node to be bound).
- **pu16InClusterList** is a pointer to the input cluster list of the local endpoint (containing clusters for matching with output clusters of remote node).
- **u8NumOutClusters** is the number of output clusters of the local endpoint (available for matching with input clusters of remote node to be bound).
- **pu16OutClusterList** is 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\_tsAplZdpBindUnbindReq structure 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:

- **u64SrcAddress** is the IEEE address of the source node for the binding
- **u8SrcEndpoint** is the number of the source endpoint for the binding (1-240)
- **u16ClusterId** is the ID of the cluster (on the local endpoint) for the binding

- `u8DstAddrMode` is the destination addressing mode (see Table 14 below):
  - `ZPS_E_ADDR_MODE_SHORT`: network address (`u8DstEndPoint` is unspecified)
  - `ZPS_E_ADDR_MODE_IEEE`: IEEE address (`u8DstEndPoint` is specified)
  - All other values are reserved
- `u16DstAddress` or `u64DstAddress` is the address of the destination node for the binding:
  - network address `u16DstAddress` if `u8DstAddrMode` is set to `ZPS_E_ADDR_MODE_SHORT`
  - IEEE address `u64DstAddress` if `u8DstAddrMode` is set to `ZPS_E_ADDR_MODE_IEEE`
- `u8DstEndPoint` is the number of the destination endpoint for the binding

(1-240) - not required if `u8DstAddrMode` set to `ZPS_E_ADDR_MODE_SHORT` (network address)

### Addressing modes

| <code>u8DstAddrMode</code> | Code                               | Description                    |
|----------------------------|------------------------------------|--------------------------------|
| 0x02                       | <code>ZPS_E_ADDR_MODE_SHORT</code> | 16-bit Network (Short) address |
| 0x03                       | <code>ZPS_E_ADDR_MODE_IEEE</code>  | 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_tsAplZdpBindRegisterReq` structure is detailed below.

```
typedef struct {
 uint64 u64NodeAddress;
} ZPS_tsAplZdpBindRegisterReq;
```

where `u64NodeAddress` is 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_tsAplZdpReplaceDeviceReq` structure is detailed below.

```
typedef struct {
 uint64 u64OldAddress; uint8 u8OldEndPoint; uint64 u64NewAddress; uint8 u8NewEndPoint;
} ZPS_tsAplZdpReplaceDeviceReq;
```

where:

- `u64OldAddress` is the IEEE address to be replaced
- `u8OldEndPoint` is the endpoint number to be replaced

(0-240, where 0 indicates that the endpoint number is not to be replaced)

- `u64NewAddress` is the replacement IEEE address
- `u8NewEndPoint` is 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:

- u64SrcAddress is the IEEE address of the source node for the binding entry
- u8SrcEndPoint is the number of the source endpoint for the binding (1-240)
- u16ClusterId is the ID of the cluster (on the local endpoint) for the binding
- u8DstAddrMode is the destination addressing mode for remaining elements (see Table 15 below)
- u16DstAddress is the address of the destination node for the binding (address type according to setting of u8DstAddrMode)
- u8DstEndPoint is 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:

- **u64SrcAddress** is the IEEE address of the source node for the binding entry.
- **u8SrcEndPoint** is the number of the source endpoint for the binding (1-240).
- **u16ClusterId** is the ID of the cluster (on the local endpoint) for the binding.
- **u8DstAddrMode** is the destination addressing mode for remaining elements (see the Table below) .
- **u16DstAddress** is the address the destination node for the binding (address type according to setting of **u8DstAddrMode**).
- **u8DstEndPoint** is 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\_tsAplZdpBackupBindTableReq structure is detailed below.

```

typedef struct {
 uint16 u16BindingTableEntries;
 uint16 u16StartIndex;
 uint16 u16BindingTableListCount;
 /* Rest of message is variable length */
 ZPS_tsAplZdpBindingTable sBindingTable;
} ZPS_tsAplZdpBackupBindTableReq;

```

where:

- **u16BindingTableEntries** is the total number of entries in the primary binding table cache.
- **u16StartIndex** is the binding table index of the first entry to be backed up.
- **u16BindingTableListCount** is the number of binding table entries in the list to be backed up (**sBindingTable**).
- **sBindingTable** is a pointer to the list of binding table entries to be backed up. Each list item is of the type **ZPS\_tsAplZdpBindingTable** detailed below:

### ZPS\_tsAplZdpBindingTable

```

typedef struct
{
 uint64 u64SourceAddress;

```

(continues on next page)

(continued from previous page)

```
ZPS_tsAplZdpBindingTableEntry* psBindingTableEntryForSpSrcAddr;
}ZPS_tsAplZdpBindingTable;
```

where:

- `u64SourceAddress` is the IEEE source address for the binding table entry.
- `psBindingTableEntryForSpSrcAddr` is the binding table entry. This is of the type `ZPS_tsAplZdpBindingTableEntry` detailed 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:

- `u16ClusterId` is the ID of the cluster (on the local endpoint) for the binding
- `u8SrcEndpoint` is the number of the source endpoint for the binding (1-240)
- `u8DstAddrMode` is the destination addressing mode for remaining elements (see Table below)
- `u16DstAddress` is the address the destination node for the binding (address type according to setting of `u8DstAddrMode`)
- `u8DstEndPoint` is the number of the destination endpoint for the binding (1-240)

### **Addressing modes**

| <b>u8DstAddrMode</b> | <b>Code</b>                        | <b>Description</b>      |
|----------------------|------------------------------------|-------------------------|
| 0x01                 | <code>ZPS_E_ADDR_MODE_GROUP</code> | 16-bit Group address    |
| 0x03                 | <code>ZPS_E_ADDR_MODE_IEEE</code>  | 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_tsAplZdpRecoverBindTableReq` structure is detailed below.

```
typedef struct {
 uint16 u16StartIndex;
} ZPS_tsAplZdpRecoverBindTableReq;
```



where `u16StartIndex` is 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_tsAplZdpBackupSourceBindReq` structure is detailed below.

```
typedef struct {
 uint16 u16SourceTableEntries;
 uint16 u16StartIndex;
 uint16 u16SourceTableListCount;
 /* Rest of message is variable length */
 uint64* pu64SourceAddress;
} ZPS_tsAplZdpBackupSourceBindReq;
```

where:

- `u16SourceTableEntries` is the total number of entries in the source binding table.
- `u16StartIndex` is the binding table index of the first entry to be backed up.
- `u16SourceTableListCount` is the number of binding table entries in the list to be backed up (`pu64SourceAddress`).
- `pu64SourceAddress` is 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_tsAplZdpRecoverSourceBindReq` structure is detailed below.

```
typedef struct {
 uint16 u16StartIndex;
} ZPS_tsAplZdpRecoverSourceBindReq;
```

where `u16StartIndex` is 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_tsAplZdpMgmtNwkDiscReq` structure is detailed below.

```
typedef struct {
 uint32 u32ScanChannels;
 uint8 u8ScanDuration;
 uint8 u8StartIndex;
} ZPS_tsAplZdpMgmtNwkDiscReq;
```

where:

- `u32ScanChannels` is 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
- `u8ScanDuration` is a value in the range 0x00 to 0x0E that determines the time spent scanning each channel - this time is proportional to  $2 \times \text{u8ScanDuration} + 1$
- `u8StartIndex` is 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\_tsAplZdpMgmtLqiReq structure is detailed below.

```
typedef struct {
 uint8 u8StartIndex;
} ZPS_tsAplZdpMgmtLqiReq;
```

where `u8StartIndex` is 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\_tsAplZdpMgmtRtgReq structure is detailed below.

```
typedef struct {
 uint8 u8StartIndex;
} ZPS_tsAplZdpMgmtRtgReq;
```

where `u8StartIndex` is 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\_tsAplZdpMgmtBindReq structure is detailed below.

```
typedef struct {
 uint8 u8StartIndex;
} ZPS_tsAplZdpMgmtBindReq;
```

where `u8StartIndex` is 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\_tsAplZdpMgmtLeaveReq structure is detailed below.

```
typedef struct {
 uint64 u64DeviceAddress;
 uint8 u8Flags;
} ZPS_tsAplZdpMgmtLeaveReq;
```

where:

- u64DeviceAddress is the IEEE address of the device being asked to leave the network.
- u8Flags is 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\_tsAplZdpMgmtDirectJoinReq structure is detailed below.

```
typedef struct { uint64 u64DeviceAddress; uint8 u8Capability; } ZPS_tsAplZdpMgmtDirectJoinReq;
```

where:

- u64DeviceAddress is the IEEE address of the device to be allowed to join
- u8Capability is 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\_tsAplZdpMgmtPermitJoiningReq structure is detailed below.

```
typedef struct {
 uint8 u8PermitDuration; bool_t bTcSignificance;
} ZPS_tsAplZdpMgmtPermitJoiningReq;
```

where:

- u8PermitDuration is the time period, in seconds, during which joining will be allowed (0x00 means that joining is enabled or disabled with no time limit)
- bTcSignificance determines 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
- u16NwkManagerAddr is 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\_tsAplZdpNwkAddrRsp structure 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:

- u8Status is the return status for **ZPS\_eAplZdpNwkAddrRequest()**
- u64IeeeAddrRemoteDev is the IEEE address of the remote node that sent the response (this is the IEEE address specified in the original request)
- u16NwkAddrRemoteDev is 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)
- `u8StartIndex` is 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` is 0.
- `pNwkAddrAssocDevList` is 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` is 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\_tsAplZdpIeeeAddrRsp structure 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:

- `u8Status` is the return status for **ZPS\_eAplZdpIeeeAddrRequest()**.
- `u64IeeeAddrRemoteDevis` is the IEEE address of the remote node that sent the response (this is the IEEE address that was requested).
- `u16NwkAddrRemoteDevis` is 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).
- `u8StartIndex` is 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` is 0.
- `pNwkAddrAssocDevList` is 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` is 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\_tsAplZdpNodeDescRsp structure 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).

- `u8Length` is the length of the returned Complex descriptor, in bytes.
- `sComplexDescriptor` is the returned Complex descriptor, a structure of type `ZPS_tsAplZdpComplexDescRsp` (described below). This is only included if `u8Status` reports success.

### **ZPS\_tsAplZdpComplexDescElement**

```
typedef struct {
 uint8 u8XMLTag;
 uint8 u8FieldCount;
 uint8 *pu8Data;
} ZPS_tsAplZdpComplexDescElement;
```

where:

- `u8XMLTag` is the XML tag for the current field.
- `u8FieldCount` is the number of fields in the Complex descriptor.
- `*pu8Data` is 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_tsAplZdpUserDescRsp` structure 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:

- `u8Status` is the return status for **`ZPS_eAplZdpUserDescRequest()`**.
- `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 User descriptor, in bytes (maximum: 16).
- `szUserDescriptor` is the returned User descriptor as a character array. This is only included if `u8Status` reports 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_tsAplZdpMatchDescRsp` structure 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()**.
- `u16NwkAddr` 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_tsAplZdpPowerDescStoreRsp` structure is detailed below.

```
typedef struct {
 uint8 u8Status;
 uint64 u64IeeeAddr;
 /* Rest of message is variable length */
 ZPS_tsAplZdpNodePowerDescriptor sPowerDescriptor;
} ZPS_tsAplZdpPowerDescStoreRsp;
```

where:

- `u8Status` is the return status for **ZPS\_eAplZdpPowerDescStoreRequest()**.
- `u64IeeeAddr` is the IEEE/MAC address of the device whose Power descriptor has been stored in the primary discovery cache.
- `sPowerDescriptor` is 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_tsAplZdpSimpleDescStoreRsp` structure is detailed below.

```
typedef struct {
 uint8 u8Status;
} ZPS_tsAplZdpSimpleDescStoreRsp;
```

where `u8Status` is 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_tsAplZdpActiveEpStoreRsp` structure is detailed below.

```
typedef struct {
 uint8 u8Status;
} ZPS_tsAplZdpActiveEpStoreRsp;
```

where `u8Status` is 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_tsAplZdpFindNodeCacheRsp` structure is detailed below.

```
typedef struct {
 uint16 u16CacheNwkAddr;
 uint16 u16NwkAddr;
 uint64 u64IeeeAddr;
} ZPS_tsAplZdpFindNodeCacheRsp;
```

where:

- u16CacheNwkAddr is the network address of the remote node that sent the response.
- u16NwkAddr is the network address of the node of interest (this is the network address that was specified in the request).
- u64IeeeAddr is 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\_tsAplZdpRemoveNodeCacheRsp structure is detailed below.

```
typedef struct {
 uint8 u8Status;
} ZPS_tsAplZdpRemoveNodeCacheRsp;
```

where u8Status is 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\_tsAplZdpEndDeviceBindRsp structure is detailed below.

```
typedef struct {
 uint8 u8Status;
} ZPS_tsAplZdpEndDeviceBindRsp;
```

where u8Status is 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\_tsAplZdpBindRsp structure is detailed below.

```
typedef struct {
 uint8 u8Status;
} ZPS_tsAplZdpBindRsp;
```

where u8Status is 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:

- **u8SourceEndpoint** is the number of the bound endpoint (1-240) on the source node of the binding
- **u16ClusterId** is the ID of the cluster involved in the binding, on the source node of the binding
- **u8DstAddrMode** is the addressing mode used in the rest of the structure (see Table 19 below)
- **u16DstAddress** is the network address of the destination node of the binding (this is only application if **u8DstAddrMode** is set to 0x03)
- **u64DstAddress** is the IEEE address of the destination node of the binding (this is only application if **u8DstAddrMode** is set to 0x04)
- **u8DstEndPoint** is 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\_tsAplZdpReplaceDeviceRsp structure is detailed below.

```
typedef struct {
 uint8 u8Status;
} ZPS_tsAplZdpReplaceDeviceRsp;
```

where **u8Status** is 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**).

- `pu64SourceTableList` is 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_tsAplZdpMgmtNwkDiscRsp` structure 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:

- `u8Status` is the return status for **ZPS\_eAplZdpMgmtNwkDiscRequest()**
- `u8NetworkCount` is the total number of networks discovered
- `u8StartIndex` is the index, in the complete list of discovered networks, of the first network reported in this response (through `psNetworkDescrList`)
- `u8NetworkListCount` is the number of discovered networks reported in this response (through `psNetworkDescrList`)
- `psNetworkDescrList` is a pointer to the first entry in a list of network descriptors for the discovered networks. Each entry is of the type `ZPS_tsAplZdpNetworkDescr` detailed below.

#### **ZPS\_tsAplZdpNetworkDescr**

```
typedef struct
{
 uint64 u64ExtPanId;
 uint8 u8LogicalChan;
 uint8 u8StackProfile;
 uint8 u8ZigBeeVersion;
 uint8 u8PermitJoining;
 uint8 u8RouterCapacity;
 uint8 u8EndDeviceCapacity;
} ZPS_tsAplZdpNetworkDescr;
```

where:

- `u64ExtPanId` is the 64-bit extended PAN ID of the discovered network.
- `u8LogicalChan` is 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).
- `u8StackProfile` is 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.
- `u8ZigBeeVersion` is the 4-bit version of the ZigBee protocol used by the discovered network.
- `u8PermitJoining` indicates 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.
- u8RouterCapacity indicates whether the device is capable of accepting join requests from Routers - set to TRUE if capable, FALSE otherwise.
  - u8EndDeviceCapacity indicates whether the device is capable of accepting join requests from End Devices - set to TRUE if 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\_tsAplZdpMgmtLqiRsp structure 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:

- u8Status is the return status for **ZPS\_eAplZdpMgmtLqiRequest()**
- u8NeighborTableEntries is the total number of Neighbor table entries on the remote node
- u8StartIndex is the Neighbor table index of the first entry reported in this response (through pNetworkTableList)
- u8NeighborTableListCount is the number of Neighbor table entries reported in this response (through pNetworkTableList)
- pNetworkTableList is a pointer to the first entry in the list of reported Neighbor table entries. Each entry is of the type ZPS\_tsAplZdpNtListEntry detailed 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:

- u64ExtPanId is the 64-bit extended PAN ID of the network .
- u64ExtendedAddress is the IEEE address of the neighboring node.
- u16NwkAddr is the network address of the neighboring node.
- u8LinkQuality is the estimated LQI (link quality) value for radio transmissions from the neighboring node.
- u8Depth is the tree depth of the neighboring node (where the Coordinator is at depth zero).
- u1Reserved1:1 is a 1-bit reserved value and should be set zero.
- u2Relationship:3 is 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:2 is 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:2 is a 2-bit value representing the ZigBee device type of the neighboring node:
  - \* 0: Coordinator
  - \* 1: Router
  - \* 2: End Device
  - \* 3: Unknown
- u6Reserved2:6 is a 6-bit reserved value and should be set zero.
- u2PermitJoining:2 is 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\_tsAplZdpMgmtRtgRsp structure 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:

- u8Status is the return status for **ZPS\_eAplZdpMgmtRtgRequest()**
- u8RoutingTableEntries is the total number of Routing table entries on the remote node
- u8StartIndex is the Routing table index of the first entry reported in this response (through pRoutingTableList)
- u8RoutingTableCount is the number of Routing table entries reported in this response (through pRoutingTableList)
- pRoutingTableList is a pointer to the first entry in the list of reported Routing table entries. Each entry is of the type ZPS\_tsAplZdpRtEntry detailed 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:

- u16NwkDstAddr is the destination network address of the route.
- u16NwkNxtHopAddr is the 'next hop' network address of the route.
- u3Status:3 is the 3-bit status for the route:
  - \* 000 = ACTIVE
  - \* 001 = DISCOVERY\_UNDERWAY
  - \* 010 = DISCOVERY\_FAILED
  - \* 011 = INACTIVE
  - \* 100 = VALIDATION\_UNDERWAY
  - \* 101-111 = Reserved.

- `u1MemConst:1` is a bit indicating whether the device is a memory-constrained concentrator.
- `u1ManyToOne:1` is a bit indicating whether the destination node is a concentrator that issued a many-to-one request.
- `u1RouteRecordReqd:1` is a bit indicating whether a route record command frame. should be sent to the destination before the next data packet.
- `u1Reserved:2` are reserved bits.
- `u8Field` contains the full set of flags of the `bfBitfieldssub`-structure, with `u3Status:3` occupying the most significant bits and `u1Reserved:2` occupying 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_tsAplZdpMgmtBindRsp` structure 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:

- `u8Status` is the return status for **ZPS\_eAplZdpMgmtBindRequest()**
- `u16BindingTableEntries` is the total number of Binding table entries on the remote node
- `u8StartIndex` is the Binding table index of the first entry reported in this response (through `sBindingTableList`)
- `u16BindingTableListCount` is the number of Binding table entries reported in this response (through `sBindingTableList`)
- `sBindingTableList` is 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_tsAplZdpMgmtLeaveRsp` structure is detailed below.

```
typedef struct {
 uint8 u8Status;
} ZPS_tsAplZdpMgmtLeaveRsp;
```

where `u8Status` is 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:

- u64ExtendedAddress is the IEEE address of the End Device.
- u16NwkAddress is 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\_tsAplZdpMgmtNwkUpdateNotify structure 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:

- u8Status is the return status for **ZPS\_eAplZdpMgmtNwkUpdateRequest()**
- u32ScannedChannels is 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
- u16TotalTransmissions is the total number of transmissions (from other networks) detected during the scan
- u16TransmissionFailures is the number of failed transmissions detected during the scan
- u8ScannedChannelListCount is the number of energy-level measurements (one per scanned channel) reported in this notification (through u8EnergyValuesList)
- u8EnergyValuesList is 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:

- pu64ChildList is a pointer to a list of 64-bit IEEE/MAC addresses of the child nodes in common.
- u8NumberOfChildren is the number of child nodes in common.
- u8Status is 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:

- `u32Length` is the size of the queue in terms of the number of messages that it can hold
- `u32ItemSize` is the size of a message, in bytes
- `u32MessageWaiting` is the number of messages currently in the queue
- `pvHead` is a pointer to the beginning of the queue storage area
- `pvWriteTo` is a pointer to the next free place in the storage area where a new message can be written
- `pvReadFrom` is 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:

- `eState` represents 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`
- `u32Time` is the remaining time, in milliseconds, that the timer still has to run
- `pvParameters` is a pointer to a set of parameters used by the timer
- `pfCallback` is 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** u32MicroIntStorage is a 32-bit mask of the interrupts currently enabled. The u32MicroIntStorage structure is used in critical sections and mutex-protected sections where u32MicroIntStorage takes 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_1     | Used as initial value in structure which receives a message collected from a message queue.                                                                                                                                                                       |
| ZPS_EVENT_4     | 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_4     | 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_4     | 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_1     | 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_1     | 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_1     | 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_1     | Indicates that network has failed to start on Coordinator.                                                                                                                                                                                                        |
| ZPS_EVENT_1     | Indicates that device failed to join network. This is reported through the structure ZPS_tsAfNwkJoinFailedEvent- see Section 8.2.2.8                                                                                                                              |
| ZPS_EVENT_1     | 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_1     | Indicates that network discovery on Router or End Device has finished and reports details of detected network through the structure ZPS_tsAfNwkDiscoveryEvent- see Section 8.2.2.9. This event (and associated structure) is generated for each network detected. |
| ZPS_EVENT_1     | 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_1     | 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_1     | Reports network status event from a remote or local node through the structure ZPS_tsAfNwkStatusIndEvent- see Section 8.2.2.13.                                                                                                                                   |
| ZPS_EVENT_1     | Indicates that a route discovery has been performed. The results are reported in the structure ZPS_tsAfNwkRouteDiscoveryConfEvent- see Section 8.2.2.14.                                                                                                          |
| ZPS_EVENT_1     | 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 Section 8.2.2.15.                                                 |
| ZPS_EVENT_1     | 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_2     | 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_2     | 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_2     | 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_1     | Indicates the results of a bound data transmission. The results are reported through the structure ZPS_tsAfBindRequestServerEvent- see Section 8.2.2.21.                                                                                                          |
| 424 ZPS_EVENT_1 | 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_4     | 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              | Value | Description                                                                                                                      |
|-------------------|-------|----------------------------------------------------------------------------------------------------------------------------------|
| ZPS_APL_ZDP_E_INV | 0x80  | 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_NOI | 0x83  | The requested endpoint is not described by a Simple descriptor.                                                                  |
| ZPS_APL_ZDP_E_NOI | 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_NOI | 0x86  | The End Device bind request was unsuccessful due to a failure to match any suitable clusters.                                    |
| ZPS_APL_ZDP_E_NOI | 0x87  | 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_NOI | 0x88  | A child descriptor was not available following a discovery request to a parent.                                                  |
| ZPS_APL_ZDP_E_INS | 0x89  | The device does not have storage space to support the requested operation.                                                       |
| ZPS_APL_ZDP_E_NOI | 0x8A  | The device is not in the proper state to support the requested operation.                                                        |
| ZPS_APL_ZDP_E_TAB | 0x8B  | The device does not have table space to support the operation.                                                                   |
| ZPS_APL_ZDP_E_NOI | 0x8C  | 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               | Value | Description                                                                                                                                       |
|--------------------|-------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| ZPS_APL_APS_E_ASD  | 0xA1  | A transmit request failed since the ASDU is too large and fragmentation is not supported.                                                         |
| ZPS_APL_APS_E_DEF  | 0xA2  | A received fragmented frame could not be defragmented at the current time.                                                                        |
| ZPS_APL_APS_E_DEF  | 0xA3  | A received fragmented frame could not be defragmented since the device does not support fragmentation.                                            |
| ZPS_APL_APS_E_ILLL | 0xA4  | A parameter value was out of range.                                                                                                               |
| ZPS_APL_APS_E_INV  | 0xA5  | An APSME-UNBIND.request failed due to the requested binding link not existing in the binding table.                                               |
| ZPS_APL_APS_E_INV  | 0xA6  | 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  | 0xA7  | A parameter value was invalid or out of range.                                                                                                    |
| ZPS_APL_APS_E_NO_  | 0xA8  | An APSDE-DATA.request requesting acknowledged transmission failed due to no acknowledgment being received.                                        |
| ZPS_APL_APS_E_NO_  | 0xA9  | 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_  | 0xAA  | 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_NOI  | 0xAB  | 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  | 0xAC  | An ASDU was received that was secured using a link key.                                                                                           |
| ZPS_APL_APS_E_SEC  | 0xAD  | An ASDU was received that was secured using a network key.                                                                                        |
| ZPS_APL_APS_E_SEC  | 0xAE  | An APSDE-DATA.request requesting security has resulted in an error during the corresponding security processing.                                  |
| ZPS_APL_APS_E_TAB  | 0xAF  | An APSME-BIND.request or APSME.ADDGROUP.request issued when the binding or group tables, respectively, were full.                                 |
| ZPS_APL_APS_E_UNSE | 0xB0  | An ASDU was received without any security.                                                                                                        |
| ZPS_APL_APS_E_UNSE | 0xB1  | 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                                 | Value | Description                                                                            |
|--------------------------------------|-------|----------------------------------------------------------------------------------------|
| ZPS_NWK_ENUM_SUCCESS                 | 0x00  | Success                                                                                |
| ZPS_NWK_ENUM_INVALID_PARAMETER       | 0xC1  | An invalid or out-of-range parameter has been passed                                   |
| ZPS_NWK_ENUM_INVALID_REQUEST         | 0xC2  | Request cannot be processed                                                            |
| ZPS_NWK_ENUM_NOT_PERMITTED           | 0xC3  | NLME-JOIN.request not permitted                                                        |
| ZPS_NWK_ENUM_STARTUP_FAILURE         | 0xC4  | NLME-NETWORK-FORMATION.request failed                                                  |
| ZPS_NWK_ENUM_ALREADY_PRESENT         | 0xC5  | NLME-DIRECT-JOIN.request failure - device already present                              |
| ZPS_NWK_ENUM_SYNC_FAILURE            | 0xC6  | NLME-SYNC.request has failed                                                           |
| ZPS_NWK_ENUM_NEIGHBOR_FAILURE        | 0xC7  | NLME-DIRECT-JOIN.request failure - no space in Router table                            |
| ZPS_NWK_ENUM_UNKNOWN_DEVICE          | 0xC8  | NLME-LEAVE.request failure - device not in Neighbor table                              |
| ZPS_NWK_ENUM_UNSUPPORTED             | 0xC9  | NLME-GET/SET.request unknown attribute identifier                                      |
| ZPS_NWK_ENUM_NO_NETWORKS             | 0xCA  | NLME-JOIN.request detected no networks                                                 |
| ZPS_NWK_ENUM_RESERVED_1              | 0xCB  | Reserved                                                                               |
| ZPS_NWK_ENUM_MAX_FRAME_COUNTER       | 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_OUTPUT          | 0xCE  | Security processing has failed on outgoing frame due CCM                               |
| ZPS_NWK_ENUM_NO_ROUTING_TABLE_SPACE  | 0xCF  | Attempt at route discovery has failed due to lack of table space                       |
| ZPS_NWK_ENUM_ROUTE_DISCOVERY_FAILURE | 0xD0  | Attempt at route discovery has failed due to any reason except lack of table space     |
| ZPS_NWK_ENUM_ROUTE_ERROR             | 0xD1  | NLDE-DATA.request has failed due to routing failure on sending device                  |
| ZPS_NWK_ENUM_BT_TABLE_FULL           | 0xD2  | Broadcast or broadcast-mode multicast has failed as there is no room in BTT            |
| ZPS_NWK_ENUM_FRAME_NOT_ROUTED        | 0xD3  | Unicast mode multi-cast frame was discarded pending route discovery                    |
| ZPS_NWK_ENUM_FRAME_IS_BUFFERED       | 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_]  | 0xE1  | CSMA/CA channel access failure                                      |
| MAC_ENUM_DENIED            | 0xE2  | GTS request denied                                                  |
| MAC_ENUM_DISABLE_TRX_FAILI | 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 NPDU (resource error) - the num       |
| ZPS_XS_E_NO_FREE_APDU                        | 0x81  | No free APDU (resource error) - the num       |
| 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 (resource)     |
| 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 table   |
| ZPS_XS_E_SIMPLE_DESCRIPTOR_NOT_FOUND | 0x88  | Simple descriptor does not exist for this entry |
| ZPS_XS_E_BAD_PARAM_APSDE_REQ_RSP     | 0x89  | Bad parameter has been found while processing   |
| 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 address |
| 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 descriptor    |
| ZPS_XS_E_PDUM_ERROR                  | 0xC5  | Decryption failed due to PDUM packet clo        |
| ZPS_XS_E_NULL_EXT_ADDR               | 0xC6  | Encryption failed due to missing the Exten      |
| 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        |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|--------------|
| <i>Default Extended PAN ID</i>  | The default Extended PAN ID (EPID) when adding 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 overlapping 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 <i>APS Use Extended PAN ID</i> — see Section 12.7. | 0             | 64 bits      |
| <i>Default Security Enabled</i> | The default setting for Security Enabled when adding new devices to the wireless network.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | true          | true / false |
| <i>Maximum number of Nodes</i>  | 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                                                                 |
|-------------------|--------------------------------------------------------------------------------------------------------------|---------------|-----------------------------------------------------------------------|
| <i>Profile Id</i> | 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)  |
| <i>Name</i>       | Textual name for the profile. This is used as a prefix for generated macro definitions in <b>zps_gen.h</b> . |               | 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              |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|--------------------|
| <i>Cluster Id</i> | 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            |
| <i>Name</i>       | Textual name for the cluster. This is used as a prefix for generated macro definitions in <b>zps_gen.h</b> .                                                                                                                        | -             | Valid C identifier |

Parent topic: [ZigBee network parameters](#)

### Coordinator parameters

| Parameter Name                              | Description                                                                                                                          | Default Value | Range              |
|---------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|---------------|--------------------|
| <b>Miscellaneous Coordinator Parameters</b> |                                                                                                                                      |               |                    |
| <i>Name</i>                                 | Textual name for the node. Used as a prefix when generating macro definitions in <b>zps_gen.h</b> .                                  |               | Valid C identifier |
| <i>Permit Time</i>                          | <i>Joining</i> Default number of seconds for which permit joining is enabled.- '255' means permanently on- '0' means permanently off | 255           | 0-255              |
| <i>Security Enabled</i>                     | Specifies whether the Coordinator will secure communication with other devices in the network.                                       | true          | true / false       |
| <i>Initial Security Key</i>                 | 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              |
|----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|--------------------|
| <b>Miscellaneous Router Parameters</b> |                                                                                                                                                                                                                                        |               |                    |
| <i>Name</i>                            | Textual name for the node. Used as a prefix when generating macro definitions in <b>zps_gen.h</b> .                                                                                                                                    |               | Valid C identifier |
| <i>Permit Joining Time</i>             | Default number of seconds for which permit joining is enabled.- 255 means permanently on- 0 means permanently off                                                                                                                      | 255           | 0-255              |
| <i>Scan Duration Time</i>              | 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             |
| <i>Security Enabled</i>                | Specifies whether the Router will secure communication with other devices in the network.                                                                                                                                              | true          | true / false       |
| <i>Initial Security Key</i>            | 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                        |
|----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|------------------------------|
| <b>Miscellaneous End Device Parameters</b>   |                                                                                                                                                                                                                                               |               |                              |
| <i>Name</i>                                  | Textual name for the node. Used as a prefix when generating macro definitions in <b>zps_gen.h</b> .                                                                                                                                           |               | Valid C identifier           |
| <i>Scan Duration Time</i>                    | The length of time to scan the selected RF channels when searching for a network to join. The time spent scanning each channel is: $(aBaseSuperframeDuration * (2n + 1))$ symbols where $n$ is the value of the Scan Duration Time parameter. | 3             | 0 – 14                       |
| <i>Security Enabled</i>                      | Specifies whether the End Device will secure communication with other devices in the network.                                                                                                                                                 | true          | true / false                 |
| <i>Initial Security Key</i>                  | The initial key that will be used when security is enabled. These are selected from the keys available on the Trust Centre.                                                                                                                   | None          |                              |
| <i>Sleeping</i>                              | 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                 |
| <i>Number of Poll Failures Before Rejoin</i> | 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                                                                            |
|-------------------------------------------------------------|----------------------------------------------------------------------------------------|
| <b>AF Parameters</b>                                        |                                                                                        |
| -                                                           |                                                                                        |
| <b>AIB Parameters</b>                                       |                                                                                        |
| APS Designated Coordinator(read only)                       | Indicates that on start-up the node is designated as the coordinator.                  |
| APS Use Extended PAN ID                                     | Indicates the Extended PAN ID (E-PAN ID) is used.                                      |
| APS Inter-frame Delay                                       | Number of milliseconds between two frames.                                             |
| APS Max Window Size                                         | APS fragmented data window size.                                                       |
| APS Non-member Radius                                       | Multicast non-member radius size.                                                      |
| APS Security Timeout Period                                 | Authentication timeout period in milliseconds.                                         |
| APS Use Insecure Join                                       | Controls action when a secured network is not found.                                   |
| <b>APS Layer Configuration Parameters</b>                   |                                                                                        |
| APS Duplicate Table Size                                    | The size of the APS layer duplicate table.                                             |
| APS Persistence Time                                        | Time, in milliseconds, for which the APS layer configuration parameters are persisted. |
| Maximum Number of Simultaneous Data Requests                | The maximum number of simultaneous data requests.                                      |
| Maximum Number of Simultaneous Data Requests with Acks      | The maximum number of simultaneous data requests with acknowledgments.                 |
| Inter PAN                                                   | True if inter PAN functionality is enabled.                                            |
| APS Poll Period                                             | The polling period, in milliseconds.                                                   |
| Maximum Number of Received Simultaneous Fragmented Messages | Maximum number of simultaneous fragmented messages.                                    |

| Parameter Name                                                            | Description                                                                                     |
|---------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| Maximum Number of Transmitted Simultaneous Fragmented Messages            | Maximum number of simultaneous fragmented messages.                                             |
| <b>Network Layer Configuration Parameters for Coordinator and Routers</b> |                                                                                                 |
| Active Neighbor Table Size                                                | Size of the active Neighbor table.                                                              |
| Child Table Size                                                          | Size of the persisted sub-table of child nodes.                                                 |
| Address Map Table Size                                                    | Size of the address map, which is used to map addresses to network IDs.                         |
| Broadcast Transaction Table Size                                          | Size of broadcast transaction table.                                                            |
| Discovery Neighbor Table Size                                             | Size of the Discovery Neighbor table.                                                           |
| Route Discovery Table Size                                                | Size of the Route Discovery table.                                                              |
| Route Record Table Size                                                   | Size of the Route Record table. If set to 0, the table is not used.                             |
| Routing Table Size                                                        | Size of the Routing table. This table is used to store the routing information for the network. |
| Security Material Sets                                                    | Number of supported network security material sets.                                             |
| <b>Network Layer Configuration Parameters for End Devices</b>             |                                                                                                 |
| Active Neighbor Table Size                                                | Size of the active Neighbor table.                                                              |
| Address Map Table Size                                                    | Size of the address map, which is used to map addresses to network IDs.                         |
| Broadcast Transaction Table Size                                          | Size of broadcast transaction table.                                                            |
| Discovery Neighbor Table Size                                             | Size of the Discovery Neighbor table.                                                           |
| 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 security material sets.                                             |
| Stack Profile                                                             | The ZigBee Stack Profile which is used to determine the network parameters.                     |

## Endpoint parameters Endpoint parameters

| Parameter Name                    | Description                                                                        | Default Value | Range              |
|-----------------------------------|------------------------------------------------------------------------------------|---------------|--------------------|
| <i>Application Device Id</i>      | Device ID for the endpoint.                                                        |               |                    |
| <i>Application Device Version</i> | Version number for the device.                                                     |               |                    |
| <i>Enabled</i>                    | Whether the endpoint is enabled or disabled.                                       | true          | true / false       |
| <i>End Point Id</i>               | The endpoint number (must be unique within the network).                           |               | 1-240              |
| <i>Name</i>                       | Textual name for the endpoint. Used as a prefix when generating macro definitions. |               | Valid C identifier |
| <i>Profile</i>                    | The application profile for the endpoint. This is 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        |
|---------------------|---------------------------------------------------------------------------------------------------------------------------|---------------|--------------|
| <i>Cluster</i>      | A link to an input cluster that will receive on the endpoint.                                                             |               |              |
| <i>Receive APDU</i> | A link to an APDU that will buffer any incoming messages.                                                                 |               |              |
| <i>Discoverable</i> | 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        |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------|---------------|--------------|
| <i>Cluster</i>        | A link to an output cluster that will transmit on the endpoint.                                                           | -             | -            |
| <i>Transmit APDUs</i> | List of APDUs that will be used to transmit the cluster.                                                                  | -             | -            |
| <i>Discoverable</i>   | Defines whether the input cluster will be present in the endpoints Simple descriptor which is used for service discovery. | 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 |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|-------|
| <i>Size</i>    | 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       |
|------------------------|-------------------------------------------------------------------------------------|---------------|-------------|
| <i>Number of NPDUs</i> | 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              |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|--------------------|
| <i>Instances</i> | The maximum number of instances of this APDU. Note that this value must be greater than the value of the parameter <i>Maximum Number of Simultaneous Data Requests with Acks</i> - see Advanced device parameters. |               |                    |
| <i>Name</i>      | 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 |
| <i>Size</i>      | 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 |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------|---------------|-------|
| <i>Size</i>    | 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        |
|---------------------------|-----------------------------------------------------------------------------------------------------------------------|-------------------------------------------|--------------|
| <i>Channel x(x=11-26)</i> | 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/<br>RT868MHz/<br>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                                                                                                                                | De-fault Value | Range        |
|--------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|----------------|--------------|
| <b>Descriptor Availability Parameters</b>        |                                                                                                                                            |                |              |
| <i>Complex Descriptor Available</i>              | Complex descriptors are not supported. Not editable.                                                                                       | false          | false        |
| <i>User Descriptor Available</i>                 | Indicates whether a user descriptor is present. Not editable.                                                                              | false          | true / false |
| <b>Descriptor Capabilities Parameters</b>        |                                                                                                                                            |                |              |
| <i>Extended Active Endpoint List Available</i>   | Indicates whether an extended active endpoint list is available. Not editable.                                                             | false          | false        |
| <i>Extended Simple Descriptor List Available</i> | Indicates whether an extended simple descriptor list is available. Not editable.                                                           | false          | false        |
| <b>MAC Capability Flags</b>                      |                                                                                                                                            |                |              |
| <i>Allocate Address</i>                          | Indicates whether the device allocates short (network) addresses or not. Not editable.                                                     |                | true / false |
| <i>Alternate PAN Coordinator</i>                 | Indicates whether the device acts as an alternative PAN Coordinator. Not editable.                                                         |                | true / false |
| <i>Device type</i>                               | Indicates whether the device is a Full Functionality Device (FFD) or Reduced Functionality Device (RFD). Not editable.                     |                | true / false |
| <i>Power source</i>                              | Indicates whether the device is mains powered or not. Not editable.                                                                        |                | true / false |
| <i>Rx On When Idle</i>                           | Indicates whether the device has its receiver enabled while the device is idle. Not editable.                                              |                | true / false |
| <i>Security</i>                                  | Indicates whether the device uses high or standard security. Only standard security is supported. Not editable.                            | false          | true / false |
| <b>Miscellaneous parameters</b>                  |                                                                                                                                            |                |              |
| <i>APS flags</i>                                 | Not editable.                                                                                                                              | 0              | 0            |
| <i>Frequency Band</i>                            | 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      |
| <i>Logical Type</i>                              | The device type: Coordinator, Router, or End Device. Not editable.                                                                         |                | ZC/ZR/ZED    |
| <i>Manufacturer Code</i>                         | The manufacturer ID code. The ZigBee Alliance allocates this code.                                                                         |                | 0 - 65535    |
| <i>Maximum buffer size</i>                       | The maximum buffer size. Not editable.                                                                                                     | 127            |              |
| <i>Maximum incoming transfer size</i>            | The maximum incoming transfer size supported by the device. This is calculated from the APDU sizes for input clusters. Not editable.       |                |              |
| <i>Maximum outgoing transfer size</i>            | The maximum incoming transfer size supported by the device. This is calculated from the APDU sizes for output clusters. Not editable.      |                |              |
| <b>System Server Capabilities parameters</b>     |                                                                                                                                            |                |              |
| <i>Backup binding</i>                            | 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                                                       |
|-------------------------------------------|---------------------------------------------------------------------|--------------------------------------|-------------------------------------------------------------|
| <b>Available Power Sources parameters</b> |                                                                     |                                      |                                                             |
| <i>Constant power</i>                     | Indicates whether a constant power source is available.             | false                                | true / false                                                |
| <i>Disposable Battery</i>                 | Indicates whether a disposable battery power source is available.   | false                                | true / false                                                |
| <i>Rechargeable Battery</i>               | Indicates whether a rechargeable battery power source is available. | false                                | true / false                                                |
| <b>Miscellaneous parameters</b>           |                                                                     |                                      |                                                             |
| <i>Default power mode</i>                 | The default power mode of the device.                               | Synchronized with RxOn-WhenIdle      | Synchronized with RxOn-WhenIdle / Periodic / Constant Power |
| <i>Default power source</i>               | 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       |
|----------------|---------------------------------------|---------------|-------------|
| <i>Size</i>    | 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   |
|---------------------|---------------------------------------|---------------|---------|
| <i>IEEE address</i> | The IEEE address to use with the key. | -             | 64 bit  |
| <i>Key</i>          | 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       |
|--------------------------|----------------------------------------------|----------------------------------------------------------------------|-------------|
| <i>Device Table Size</i> | 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 Section 12.7.9) 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 |
|--------------------|-------------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|-------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|----------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|--------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|-------------------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|---------------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|----------------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|-------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|--------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|---------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|---------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|----------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|---------------------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|-----------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|---------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|------------------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|--------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|---------------------------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|-----------------------------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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       |
|-------------------------|-------------------------------------------------------------------------------------------------------|---------------|-------------|
| <i>Output APDU</i>      | The APDU to use when replying to end device bind request messages.                                    | apduZDP       |             |
| <i>Timeout</i>          | Number of seconds before timing out an End Device bind request.                                       | 5             | 1 or higher |
| <i>Bind Num Retries</i> | 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 |
|--------------------|----------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|---------------------------------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | 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 |
|--------------------|-------------------------------------------------------------|---------------|-------|
| <i>Output APDU</i> | The APDU to use when replying to MIB IEEE request messages. | apduZDP       |       |

**Parent topic:** [ZigBee network parameters](#)

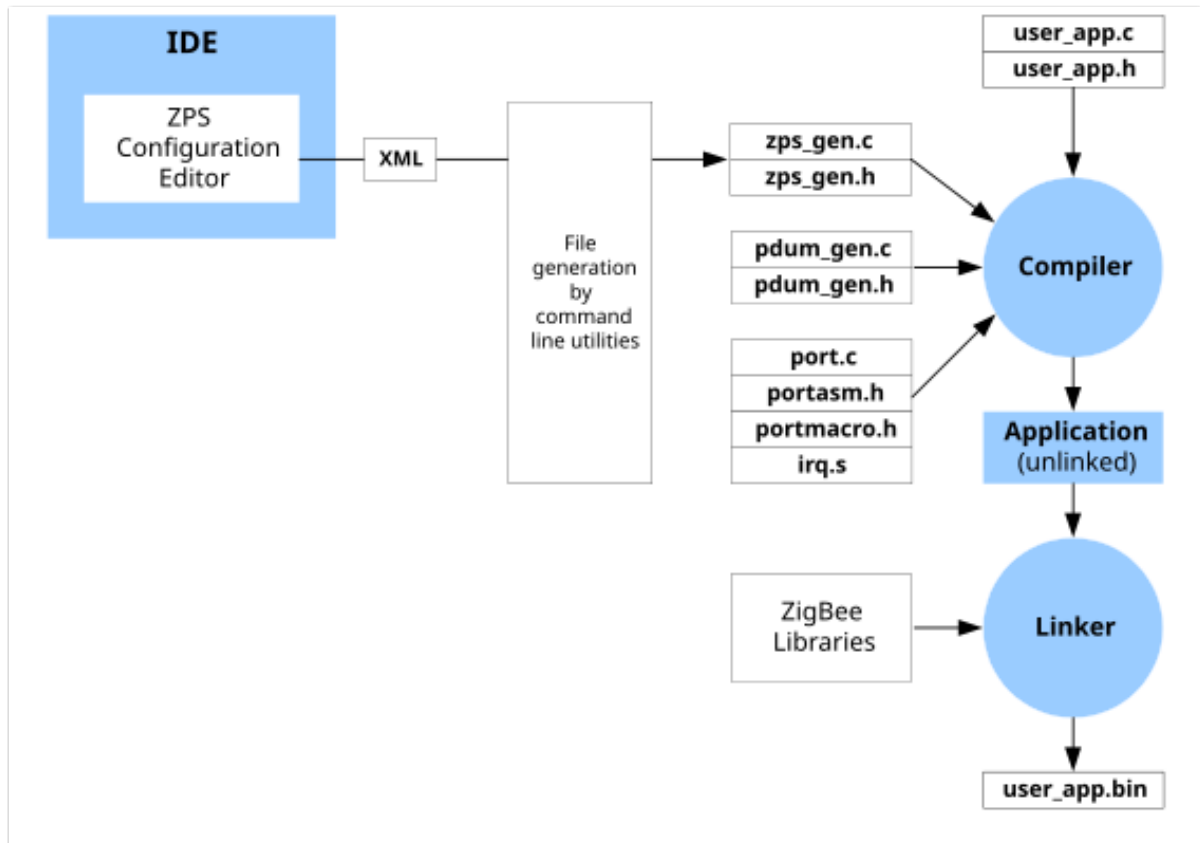
**ZPS Configuration Editor** In developing a ZigBee PRO application, certain static configuration is required before the application is built. The ZPS Configuration Editor is used to simplify this configuration. This editor is supplied as an NXP plug-in for Eclipse and is provided in the ZigBee 3.0 SDK (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



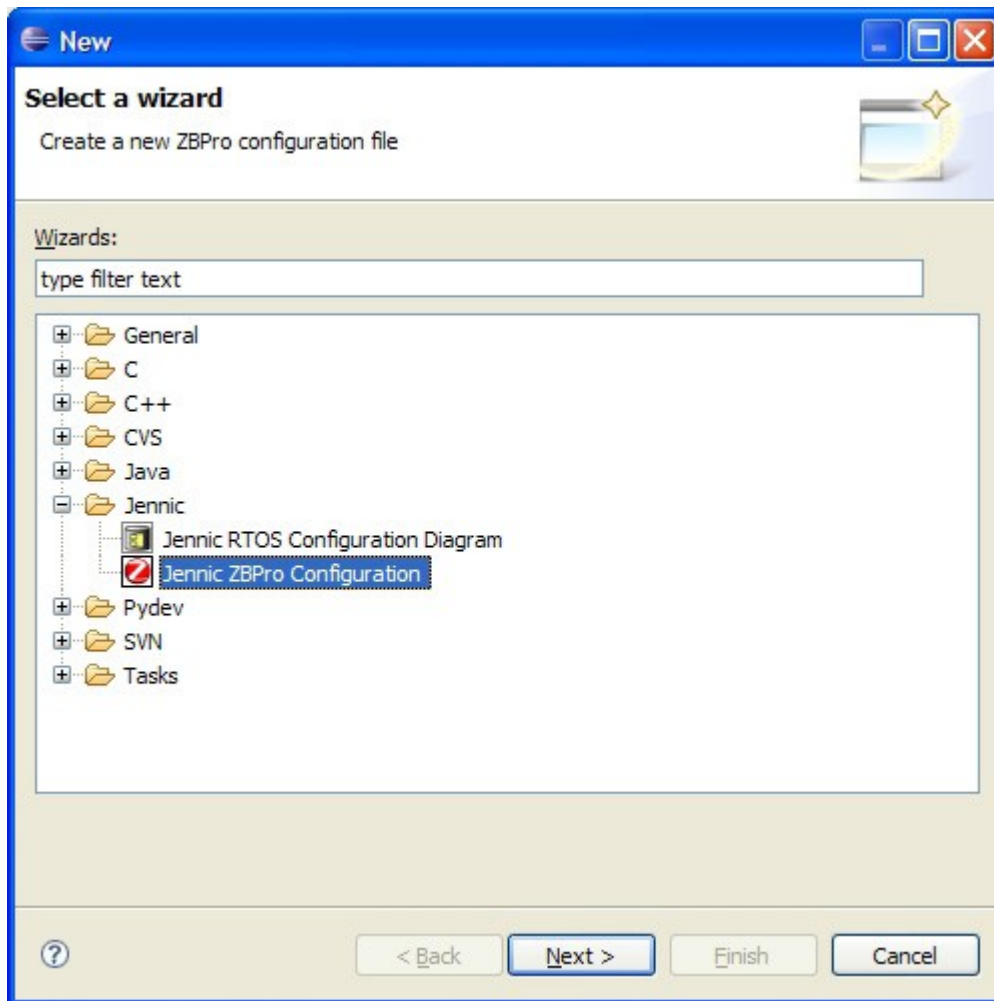
Parent topic: [ZPS Configuration Editor](#)

**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.

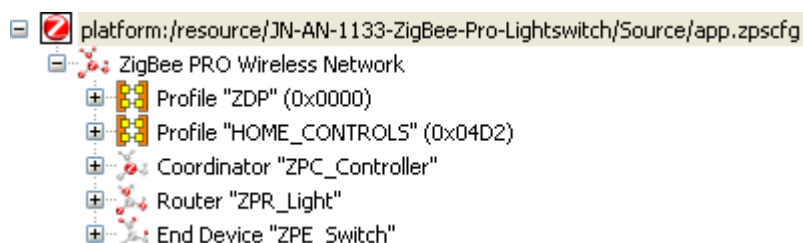
**Parent topic:** [ZPS Configuration Editor](#)

**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:

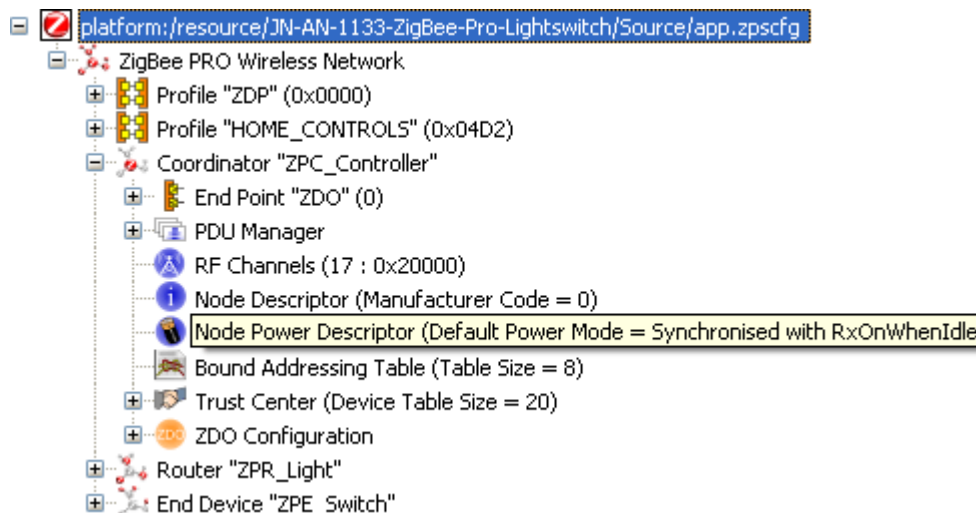
- Entries for the ZigBee application profiles used in the network
- Entry for the Coordinator
- Entries for the Routers
- Entries for the End Devices

**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

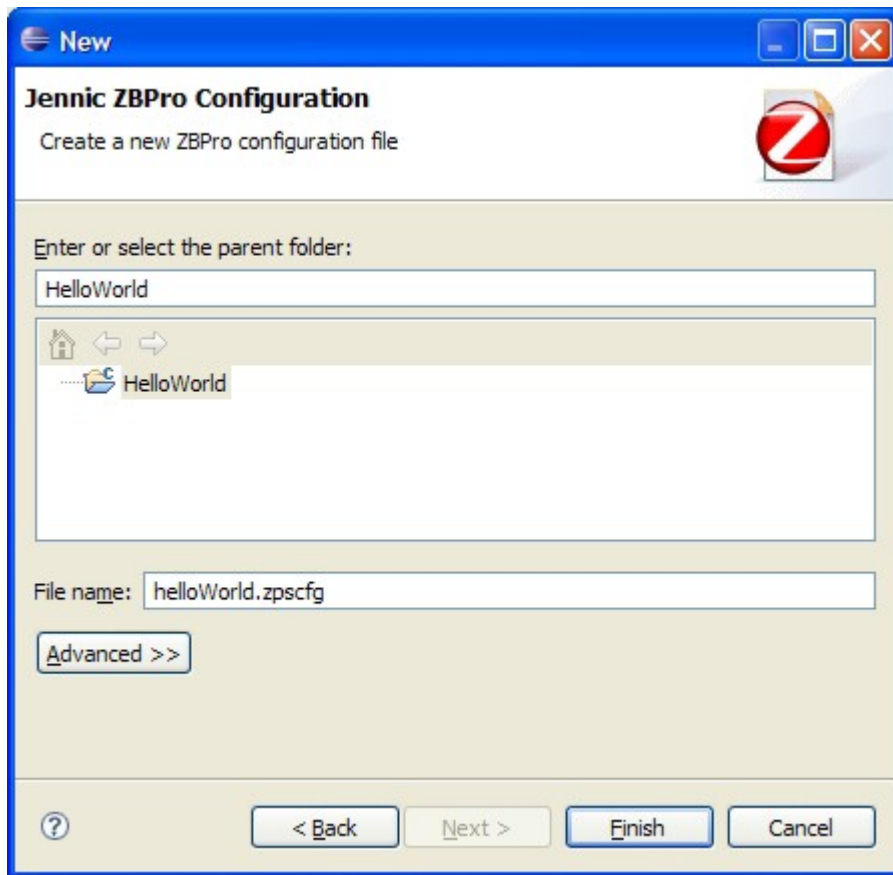
**Parent topic:** [ZPS Configuration Editor](#)

**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.

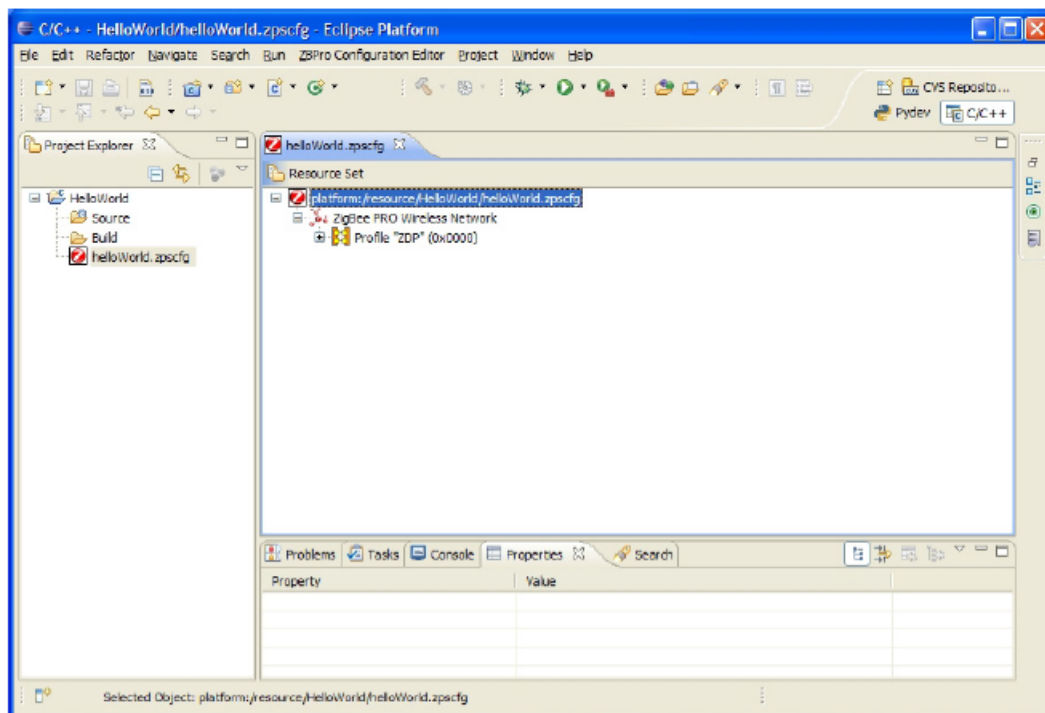
**New ZPS Configuration**



**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.

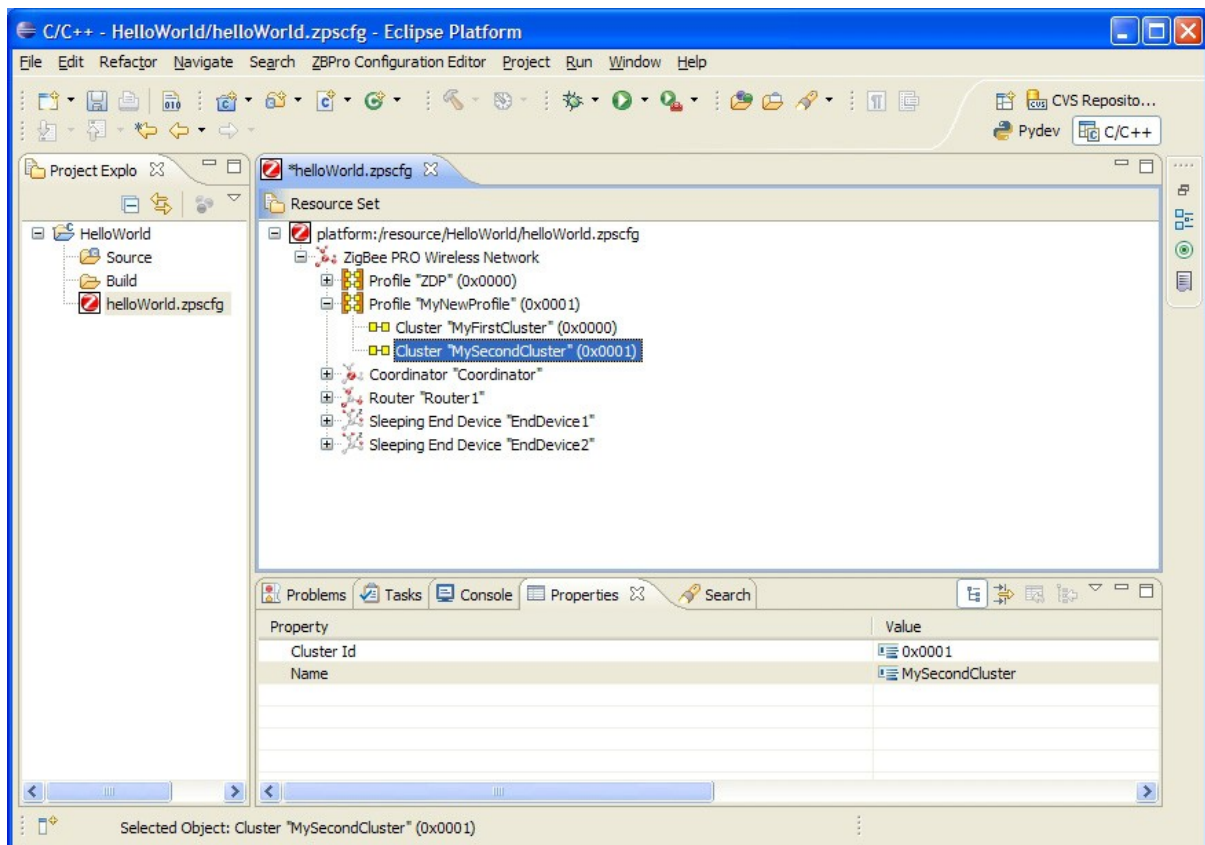
**Step 2** Edit the properties in the **Properties** tab to set **Name** and **Id** for the new profile.

**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:

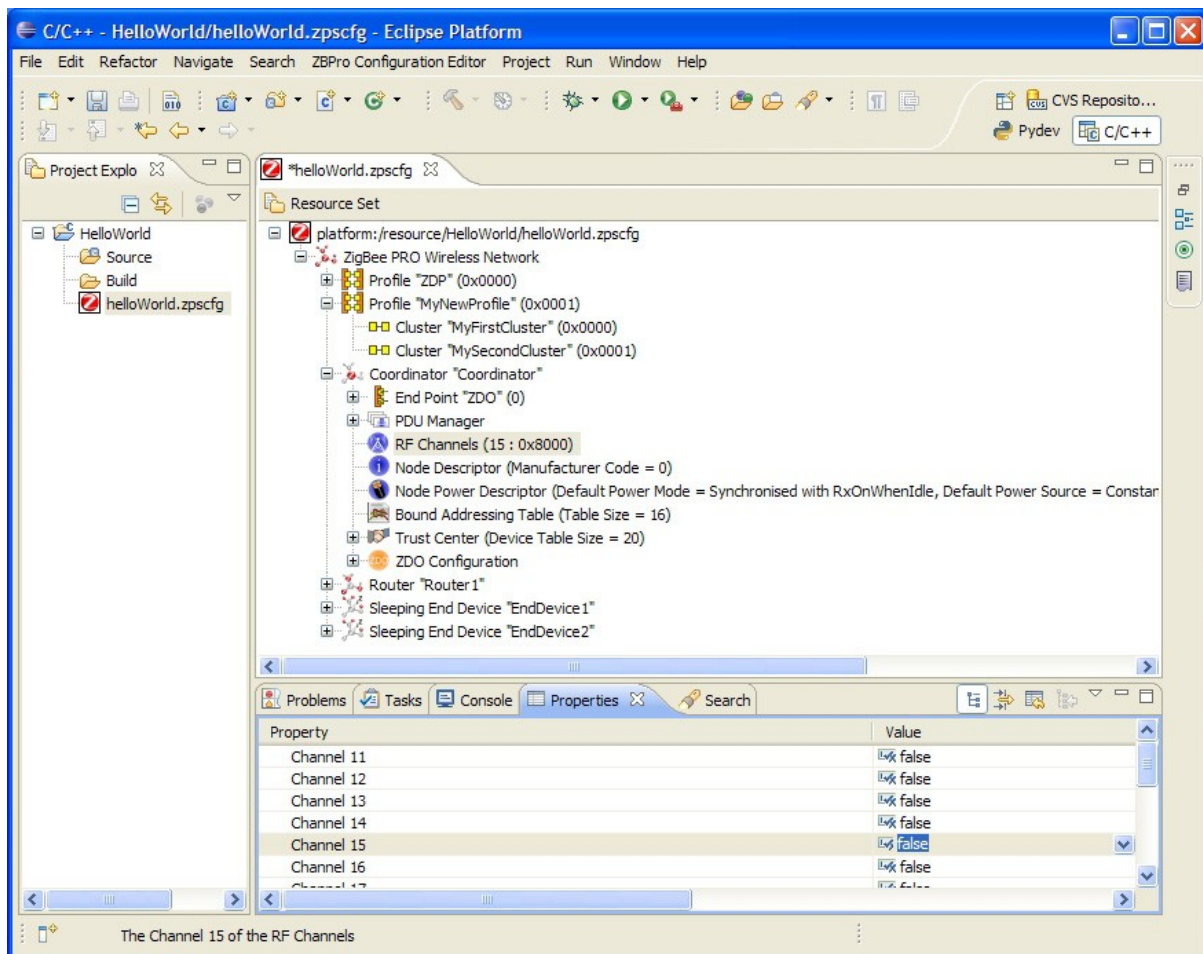
**Step 1:** Expand the Coordinator node in the editor. This reveals the default set of features for the Coordinator, ZDO endpoint, and ZDO servers.

**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.

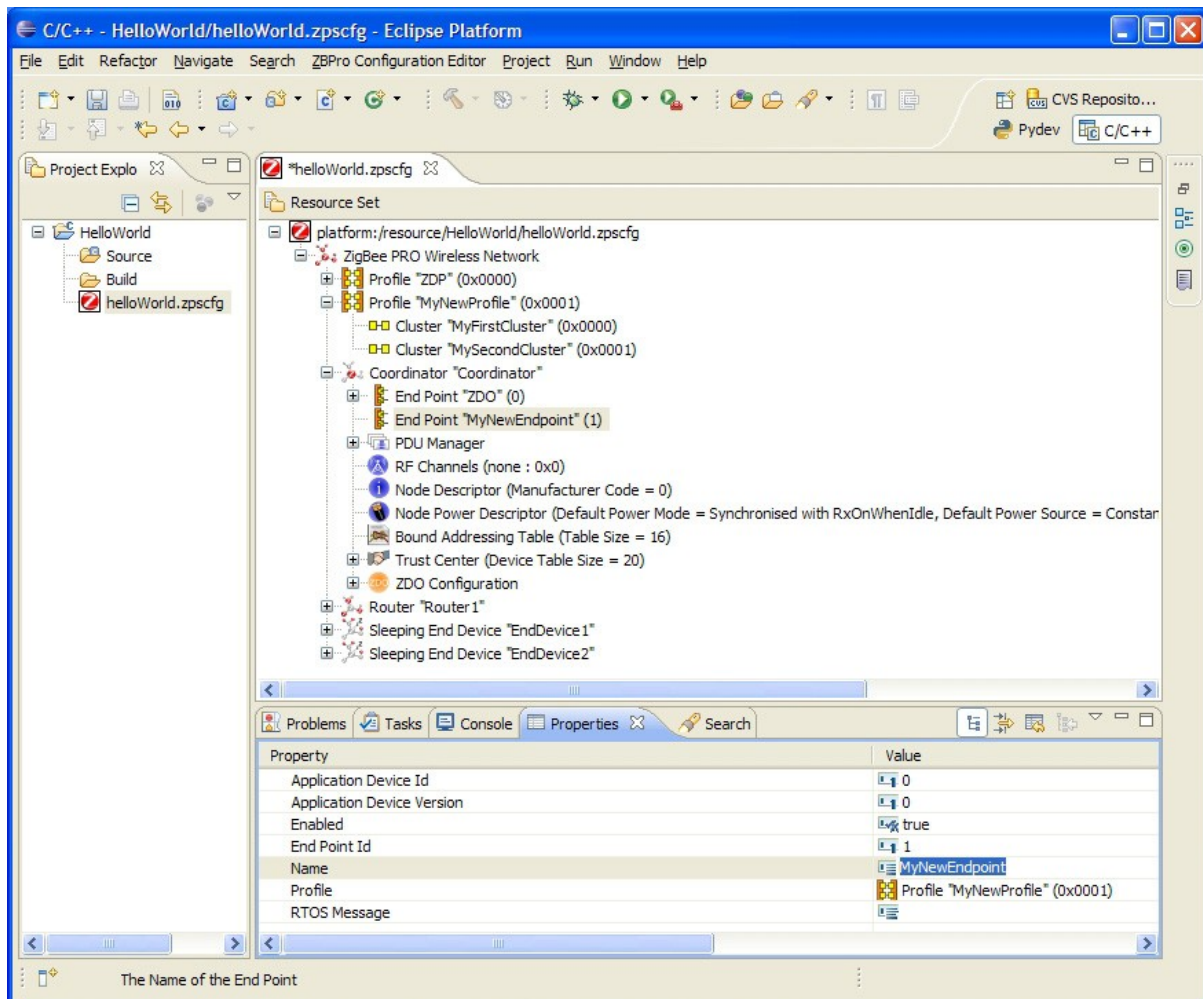
**To add a new endpoint** **Step 1:** Right-click on the Coordinator node and select **New Child > End Point** from the drop-down menu.

**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



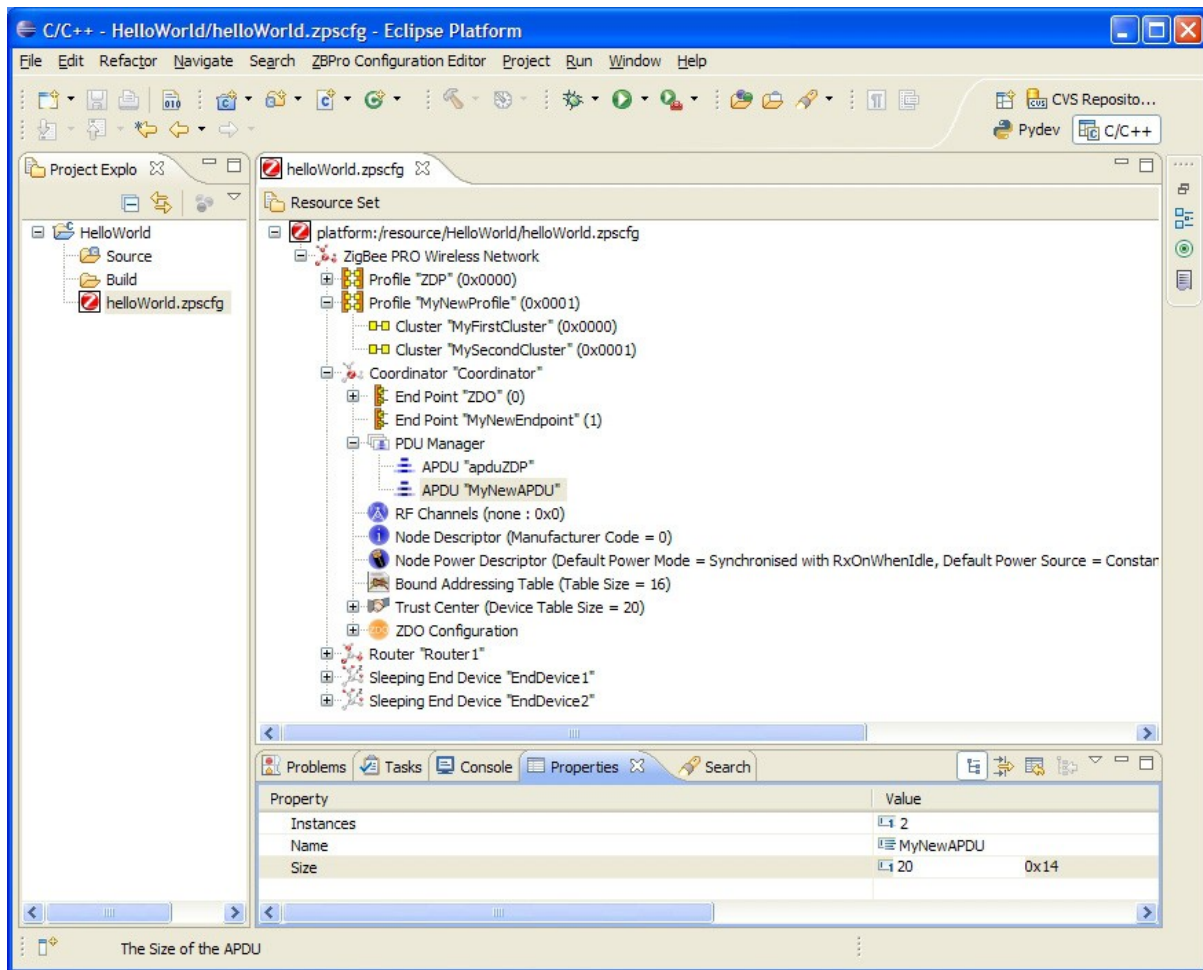


**Parent topic:**Setting Coordinator properties

**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.

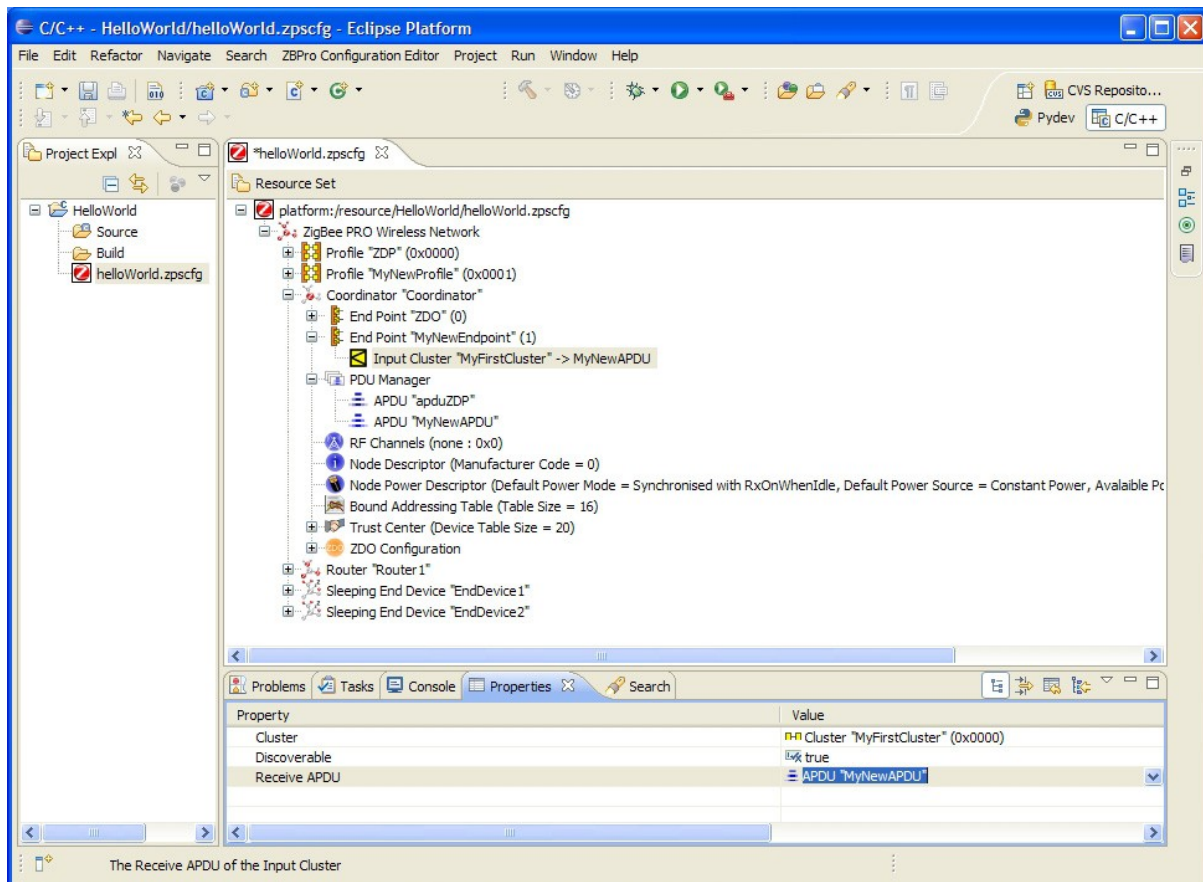
**Step 2:** Edit the properties in the **Properties** tab to set **Cluster**- select from the available clusters in the drop-down list.

**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.

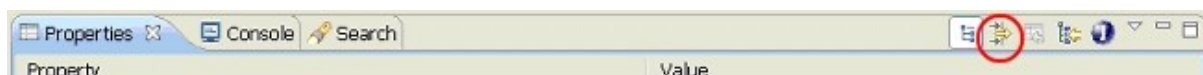
**Parent topic:** Setting Coordinator properties

**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 re-assembled.

**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 `tsBeaconFilterType` structure 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 `u8VsFormEdThreshold` is `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 `u8VsFormEdThreshold` to 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 `u8VsFormEdThreshold` and 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 ha      |
| 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 anothe      |
| 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 i     |
| 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 Coord     |
| End Device                   | A node which has no networking role (such as routing) and is only concerned v      |
| 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 transi       |
| 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 mess        |
| PAN ID                       | Personal Area Network Identifier: This is a 16-bit value that uniquely identifie   |
| 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 r |
| 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 ar       |
| 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.

| Docu-<br>ment ID | Release<br>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 K32WJN5189                                                                      |
| 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.

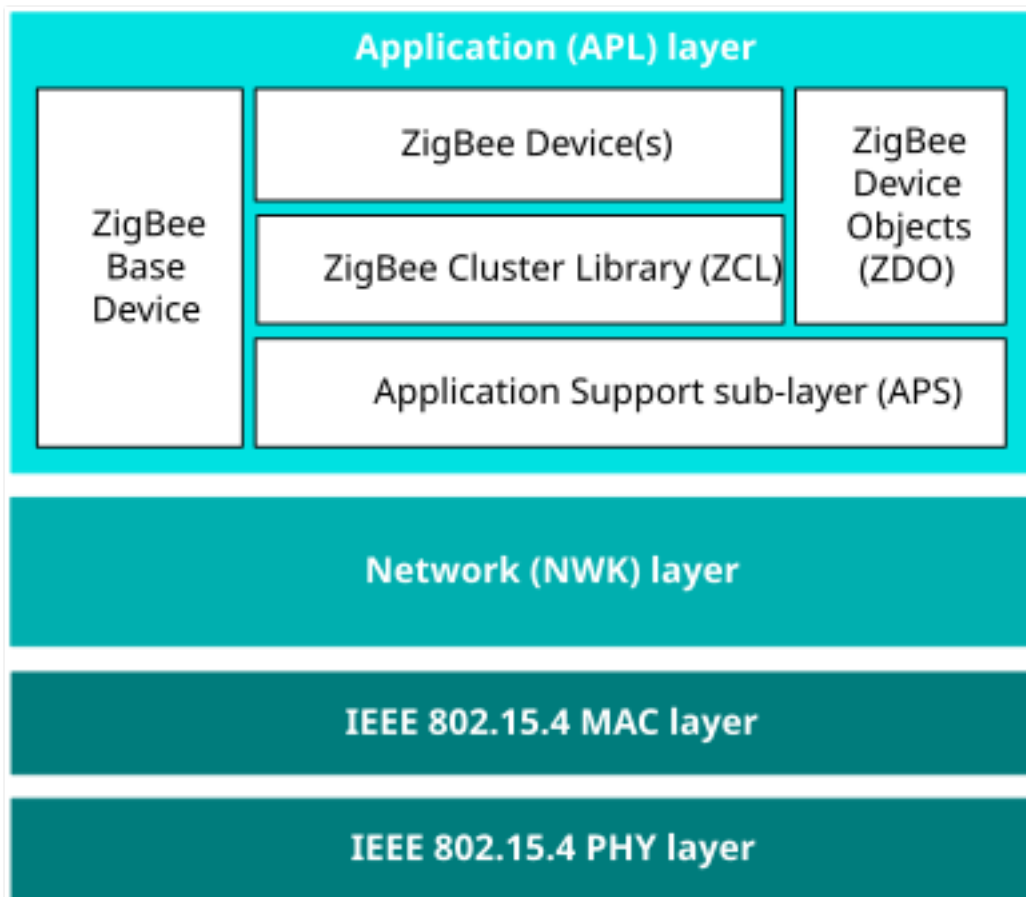
In addition, every ZigBee 3.0 node must employ the following devices:

- **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.

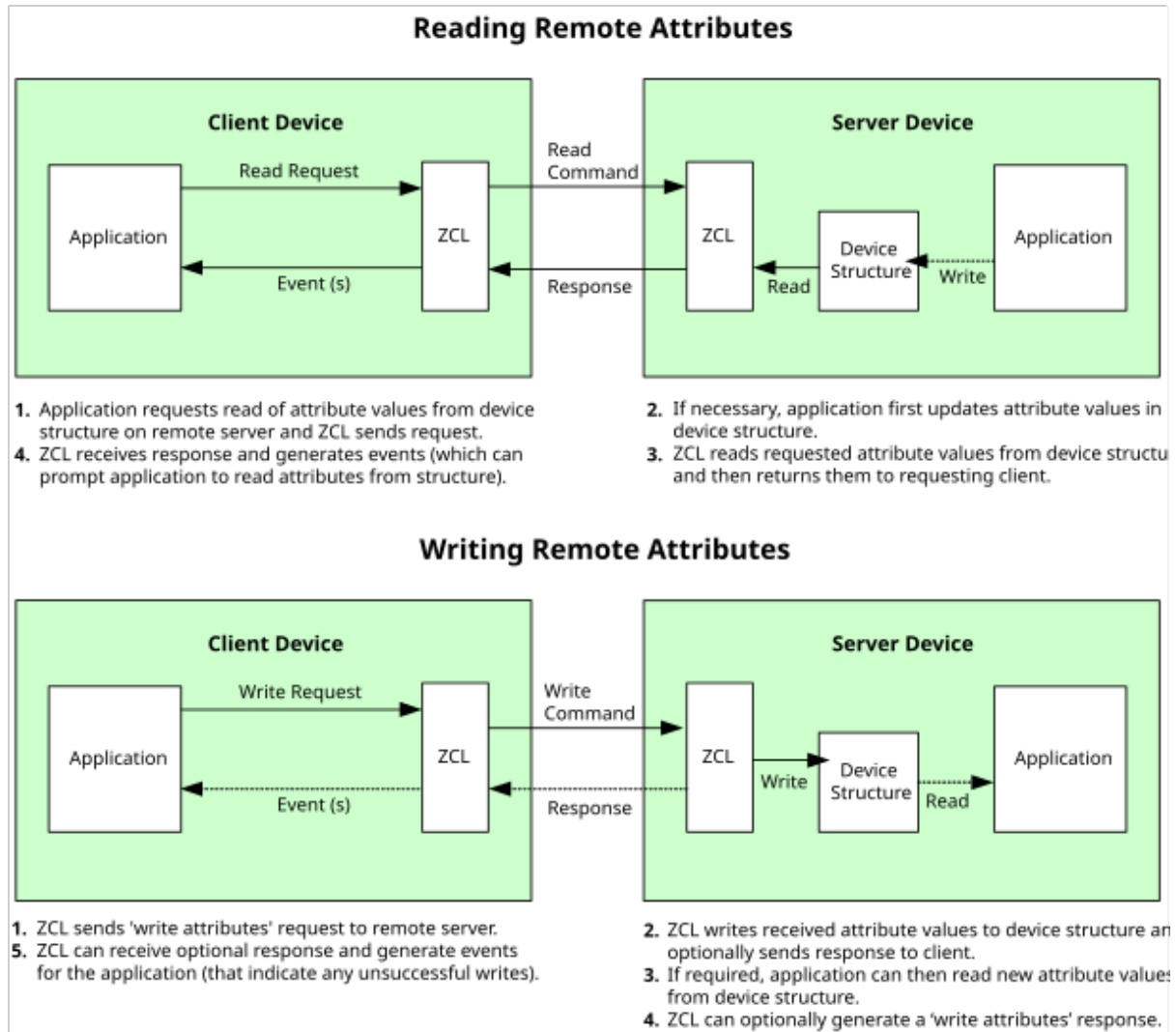
**Note:** In order to use a cluster which is supported by a device, the relevant option for the cluster must be specified at build-time - see Compile-time options.

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\_eAplAfnit()**, 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\_eAplAfnit()**, 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_SUCCESS` event.

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           | <ul style="list-style-type: none"> <li>• Creating a new network</li> <li>• Allowing other devices to join an existing network</li> </ul>                      |
| Network steering    | <ul style="list-style-type: none"> <li>• Allowing other devices to join an existing network</li> <li>• Joining local device to an existing network</li> </ul> |
| Network formation   | <ul style="list-style-type: none"> <li>• Creating a new network</li> </ul>                                                                                    |
| Finding and binding | <ul style="list-style-type: none"> <li>• Binding a local endpoint to an endpoint on a remote node</li> <li>• Adding a remote node to a group</li> </ul>       |

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))*[\[https://kex-daily.nxp.com/mcuxsdk-internal/latest/html/\\_static/wireless/Zigbee/JN-UG-3132-ZigBee3-Cluster-Library.pdf\]](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:

```
typedef struct
{
 uint16 u16bdbCommissioningGroupID;
 uint8 u8bdbCommissioningMode;
 BDB_teCommissioningStatus u8bdbCommissioningStatus;
 uint64 u64bdbJoiningNodeEui64;
 uint8 au8bdbJoiningNodeNewTCLinkKey[16];
 bool_t bbdbJoinUsesInstallCodeKey;
 const uint8 u8bdbNodeCommissioningCapability;
 bool_t bbdbNodeIsOnANetwork;
 uint8 u8bdbNodeJoinLinkKeyType;
 uint32 u32bdbPrimaryChannelSet;
 uint8 u8bdbScanDuration;
 uint32 u32bdbSecondaryChannelSet;
 uint8 u8bdbTCLinkKeyExchangeAttempts;
 uint8 u8bdbTCLinkKeyExchangeAttemptsMax;
 uint8 u8bdbTCLinkKeyExchangeMethod;
 uint8 u8bdbTrustCenterNodeJoinTimeout;
 bool_t bbdbTrustCenterRequireKeyExchange;
 bool_t bTLStealNotAllowed;
 bool_t bLeaveRequested;
} 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: `bbdbCommissioningCapability` 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_TARGET |
| 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:

$$aBaseSuperframeDuration \times (2bdbScanDuration + 1)$$

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 be 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 Co-ordinator/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                                             |
|----------------------------------------------|---------------------------------------------------|
| <code>bdbcMaxSameNetworkRetryAttempts</code> | <code>BDBC_MAX_SAME_NETWORK_RETRY_ATTEMPTS</code> |
| <code>bdbcMinCommissioningTime</code>        | <code>BDBC_MIN_COMMISSIONING_TIME</code>          |
| <code>bdbcRecSameNetworkRetryAttempts</code> | <code>BDBC_REC_SAME_NETWORK_RETRY_ATTEMPTS</code> |
| <code>bdbcTCLinkKeyExchangeTimeout</code>    | <code>BDBC_TC_LINK_KEY_EXCHANGE_TIMEOUT</code>    |

**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                                           |
| <code>bdbcTLInterPANTransIdLifetime</code> | <code>BDBC_TL_INTERPAN_TRANS_ID_LIFETIME</code> |
| <code>bdbcTLMinStartupDelayTime</code>     | <code>BDBC_TL_MIN_STARTUP_DELAY_TIME</code>     |
| <code>bdbcTLPrimaryChannelSet</code>       | <code>BDBC_TL_PRIMARY_CHANNEL_SET</code>        |
| <code>bdbcTLRxWindowDuration</code>        | <code>BDBC_TL_RX_WINDOW_DURATION</code>         |
| <code>bdbcTLScanTimeBaseDuration</code>    | <code>BDBC_TL_SCAN_TIME_BASE_DURATION_MS</code> |
| <code>bdbcTLSecondaryChannelSet</code>     | <code>BDBC_TL_SECONDARY_CHANNEL_SET</code>      |

**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_eAplAfnInit()`**. The ZigBee Base Device attribute *`bbdbNodeIsOnANet-work`* 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\_eAplAfnit()**. 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_t BDB_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_CAPABLE             | 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_FAILED           | Network formation failed                           |
| E_BDB_COMMISSIONING_STATUS_NO_IDENTIFY_QUERY_RESPONSE | No responses received to an Identify Query command |
| E_BDB_COMMISSIONING_STATUS_BINDING_TABLE_FULL         | The local Binding table is full                    |
| E_BDB_COMMISSIONING_STATUS_NO_SCAN_RESPONSE           | No responses received during a channel scan        |
| E_BDB_COMMISSIONING_STATUS_NOT_PERMITTED              | Requested commissioning is not permitted           |
| E_BDB_COMMISSIONING_STATUS_TCLK_EX_FAILURE            | 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 |
| <b>Mandatory</b>                    |                      |
| Basic                               |                      |
| Identify                            |                      |
| On/Off                              |                      |
| Scenes                              |                      |
| Groups                              |                      |
| <b>Optional</b>                     |                      |
| 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<br>Mandatory      | Client (Output) side |
| Basic                                 |                      |
| Identify                              |                      |
| On/Off                                |                      |
| Level control                         |                      |
| Scenes                                |                      |
| Groups                                |                      |
| <b>Optional</b>                       |                      |
| 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:

```
typedef struct
{
 tsZCL_EndPointDefinitionsEndPoint;
 /*Clusterinstances*/
 tsZLO_DimmableLightDeviceClusterInstancesClusterInstance;
 #if(definedCLD_BASIC)&&(definedBASIC_SERVER)
 /*BasicCluster-Server*/
 tsCLD_BasicsBasicServerCluster;
 #endif
 #if(definedCLD_IDENTIFY)&&(definedIDENTIFY_SERVER)
 /*IdentifyCluster-Server*/
 tsCLD_IdentifyIdentifyServerCluster;
 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 |
| <b>Mandatory</b>                            |                      |
| Basic                                       | On/Off               |
| Identify                                    | Level control        |
|                                             | Colour control       |
|                                             | Identify             |
| <b>Optional</b>                             |                      |
| 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 the table below.

| Table 17. Clusters for On/Off Light Switch |                      |
|--------------------------------------------|----------------------|
| Server (Input) side<br><b>Mandatory</b>    | Client (Output) side |
| 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;
 #endif
}
```

(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<br><b>Mandatory</b> | Client (output) side |
| Basic                                   | On/Off               |
| Identify                                | Identify             |
|                                         | Level control        |
| <b>Optional</b>                         |                      |
| 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 |
| <b>Mandatory</b>                            |                      |
| Basic                                       | On/Off               |
| Identify                                    | Level control        |
|                                             | Colour control       |
|                                             | Identify             |
| <b>Optional</b>                             |                      |
| 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 |
| <b>Mandatory</b>                    |                      |
| Basic<br>Identify                   | Identify             |
| Illuminance measurement             |                      |
| <b>Optional</b>                     |                      |
|                                     | 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 |
|---------------------|----------------------|
| <b>Mandatory</b>    |                      |
| Basic<br>Identify   | Identify             |
| Occupancy sensing   |                      |
| <b>Optional</b>     |                      |
| 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      |
|----------------------------------|---------------------------|
| <b>Mandatory</b>                 |                           |
| Basic                            |                           |
| Power configuration              |                           |
| Device temperature configuration |                           |
| Identify                         |                           |
| Groups                           |                           |
| Scenes                           |                           |
| On/Off                           |                           |
| Ballast configuration            |                           |
| <b>Optional</b>                  |                           |
| 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;
 #endif
}
```

(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      |
|----------------------------------|---------------------------|
| <b>Mandatory</b>                 |                           |
| Basic                            |                           |
| Power configuration              |                           |
| Device temperature configuration |                           |
| Identify                         |                           |
| Groups                           |                           |
| Scenes                           |                           |
| On/Off                           |                           |
| Level control                    |                           |
| Ballast configuration            |                           |
| <b>Optional</b>                  |                           |
| 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 |
| <b>Mandatory</b>                           |                      |
| Basic                                      |                      |
| Identify                                   |                      |
| Groups                                     |                      |
| Scenes                                     |                      |
| On/Off                                     |                      |
| <b>Optional</b>                            |                      |
| 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 |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------|
| <b>Mandatory</b>                                                                                                                                                |                      |
| Basic                                                                                                                                                           |                      |
| Identify                                                                                                                                                        |                      |
| Groups                                                                                                                                                          |                      |
| Scenes                                                                                                                                                          |                      |
| On/Off                                                                                                                                                          |                      |
| Level control                                                                                                                                                   |                      |
| <b>Optional</b>                                                                                                                                                 |                      |
|                                                                                                                                                                 | OTA upgrade          |
| The mandatory attributes within each cluster for this device type are indicated in the <i>ZigBee Lighting and Occupancy Device Specification (15-0014-01)</i> . |                      |

**Parent topic:**Dimmable Plug-in Unit

**Device Structure** The following `tsZLO_DimmablePlugDevice` structure is the shared structure for a Dimmable Plug-in Unit device:

```
typedef struct
{
 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<br>Mandatory | Client (Output) side |
|----------------------------------|----------------------|
| Basic                            |                      |
| Identify                         |                      |
| Groups                           |                      |
| Scenes                           |                      |
| On/Off                           |                      |
| Level control                    |                      |
| Colour control                   |                      |
| <b>Optional</b>                  |                      |
| 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 |
| <b>Mandatory</b>                             |                      |
| 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<br>Mandatory    | Client (output) side |
| Basic<br>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    |
| <b>Mandatory</b>                         |                         |
| Basic                                    | On/Off                  |
| Identify                                 | Identify                |
|                                          | Level control           |
|                                          | Colour control          |
| <b>Optional</b>                          |                         |
| 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-

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 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           |
| <b>Mandatory</b>                               |                                |
| Basic                                          | On/Off                         |
| Identify                                       | Identify                       |
|                                                | Level control                  |
|                                                | Colour control                 |
|                                                | Scenes                         |
| <b>Optional</b>                                |                                |
| 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           |
|-------------------------|--------------------------------|
| <b>Mandatory</b>        |                                |
| Basic                   | On/Off                         |
| Identify                | Identify                       |
|                         | Level control                  |
| <b>Optional</b>         |                                |
| 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                  |
| <b>Optional</b>         |                         |
| 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      |
|-------------------------|---------------------------|
| <b>Mandatory</b>        |                           |
| Basic Identify          | On/Off Identify Groups    |
|                         | Scenes                    |
|                         | Level control             |
|                         | Colour control            |
| <b>Optional</b>         |                           |
| 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<br>Mandatory | Client (Output) side                     |
|----------------------------------|------------------------------------------|
| Basic<br>Identify<br>Optional    | On/Off<br>Identify                       |
| Touchlink commissioning          | Touchlink commissioning<br>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](http://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 ID                       | Cluster ID | Description                                                                                                                                                                                                                                                                                                                                                                                                                          |
|----------------------------------|------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Basic                            | 0x00       | 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 <i>Chapter 8</i> .                                                                                                                                                                                     |
| Power Configuration              | 0x00       | 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 <i>Chapter 9</i> .                                                                                                                                                                                                                                     |
| Device Temperature Configuration | 0x00       | 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 <i>Chapter 10</i> .                                                                                                                                                                                                        |
| Identify                         | 0x00       | 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 <i>Chapter 11</i> .                                                                                                                                                                                                                            |
| Groups                           | 0x00       | 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 <i>Chapter 12</i> .                                                                                                                                                                                                                   |
| Scenes                           | 0x00       | 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 <i>Chapter 13</i> .                                                                                                                                                                           |
| On/Off                           | 0x00       | 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 <i>Chapter 14</i> .                                                                                                                                                                                                                                                                      |
| On/Off Switch Configuration      | 0x00       | 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 <i>Chapter 15</i> .                                                                                                                                                                                                   |
| Level Control                    | 0x00       | 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 <i>Chapter 16</i> .                                                                                                                                                                                                                                                                 |
| Alarms                           | 0x00       | 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 <i>Chapter 17</i> .                                                                                                                                                                  |
| Time                             | 0x00       | 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 <i>Chapter 18</i> . |
| Analogue Input (Basic)           | 0x00       | The Analogue Input (Basic) cluster provides an interface for accessing an analog measurement. This cluster is detailed in Section 19.1.                                                                                                                                                                                                                                                                                              |
| Analogue Output (Basic)          | 0x00       | 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 (Basic)             | 0x00       | 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 <i>Chapter 23</i> .                     |
| Illuminance Level Sensing     | 0x04       | The Illuminance Level Sensing cluster provides an interface to light-level sensing functionality. This cluster is detailed in <i>Chapter 24</i> .                                                                                            |
| 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 <i>Chapter 25</i> .                      |
| Pressure Measurement          | 0x04       | The Pressure Measurement cluster provides an interface to a pressure measuring device, allowing the configuration of measuring and the reporting of measurements. This cluster is detailed in <i>Chapter 26</i> .                            |
| 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 <i>Chapter 27</i> .                        |
| 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 <i>Chapter 28</i> . |
| 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 <i>Chapter 29</i> .                                               |
| Electrical Measurement        | 0x0E       | The Electrical Measurement cluster provides an interface for obtaining electrical measurements from a device. This cluster is detailed in <i>Chapter 30</i> .                                                                                |

**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 <i>Chapter 31</i> . |
| 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 <i>Chapter 32</i> .                                             |

**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 <i>Chapter 33</i> .                                                                                  |
| Fan Control                             | 0x02       | The Fan Control cluster provides a means of controlling the speed or state of a fan which may be part of a heating or cooling system. The cluster is detailed in <i>Chapter 34</i> .                                                       |
| Thermostat User Interface Configuration | 0x02       | The Thermostat User Interface (UI) Configuration cluster provides a means of configuring the user interface (keypad and/or LCD screen) for a thermostat or a thermostat controller device. This cluster is detailed in <i>Chapter 35</i> . |

**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 <i>Chapter 36</i> . |

**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 <i>Chapter 37</i> .                                                                                                                                                                               |
| 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 <i>Chapter 38</i> .                                                                                                                                       |
| 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 <i>Chapter 39</i> . |

**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 <i>Chapter 40</i> .                                                                                                                                     |
| 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 <i>Chapter 41</i> . |
| Simple Metering                  | 0x07       | The Simple Metering cluster provides a mechanism to obtain consumption data from a metering device (electric, gas, water or thermal). This cluster is detailed in <i>Chapter 42</i> .                                                                                                                   |

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 <i>Chapter 43</i> . |
| 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 <i>Chapter 44</i> .                                                                                                                        |

**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 <i>Chapter 45</i> .                           |
| Appliance Identification    | 0x0B0      | The Appliance Identification cluster provides an interface for obtaining and setting basic appliance information. This cluster is detailed in <i>Chapter 46</i> .              |
| 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 <i>Chapter 47</i> . |
| Appliance Statistics        | 0x0B0      | The Appliance Statistics cluster provides an interface for supplying statistical information about an appliance. This cluster is detailed in <i>Chapter 48</i> .               |

**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 Upgrade | 0x0C       | The OTA Upgrade cluster provides the facility to upgrade (or downgrade or re-install) 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 <i>Chapter 49</i> . |

**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: CommonResources*:

- *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:

```
#undef 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](#)

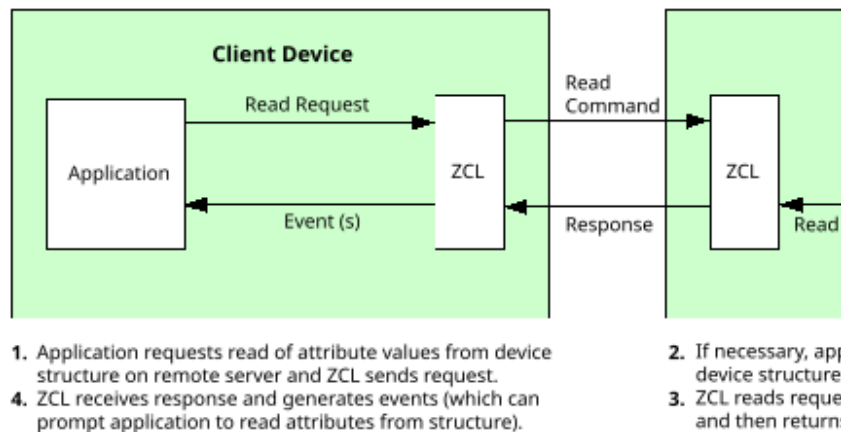
**Shared Device Structures** In each ZigBee device, cluster attribute values are exchanged between the application and the ZCL by means of a shared structure. This structure is protected by a mutex - see *Appendix A*. The structure for a particular ZigBee device contains structures for the clusters supported 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.

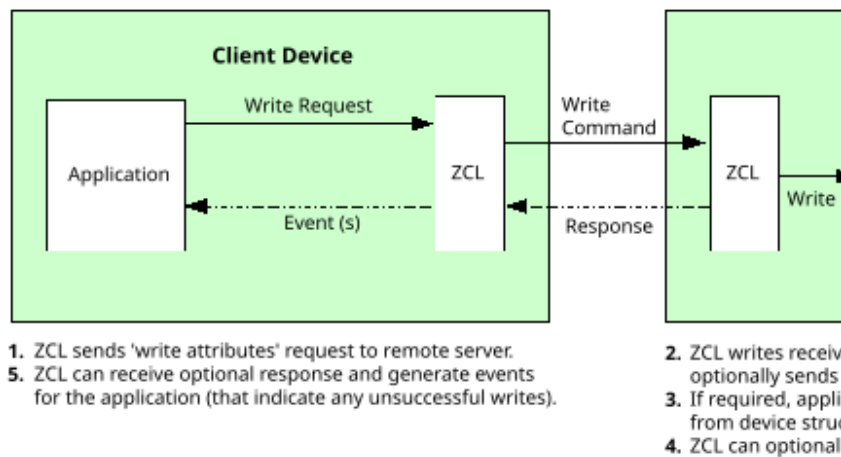
A shared device structure within a device can be accessed both by the local application and by a remote application on another device. Remote read and write operations involving a shared device structure are illustrated in the figure below. Normally, a cluster client requests these operations and they 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.

## Reading Remote Attributes



## Writing Remote Attributes



## 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)(amp((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)(amp((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)(amp((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)(amp((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)(amp((tsCLD_OnOff*)(0))->eStartUpOnOff),0}, /* Optional */
#endif
#ifdef
 {E_CLD_GLOBAL_ATTR_ID_CLUSTER_REVISION, (E_ZCL_AF_RD|E_ZCL_AF_GA), E_ZCL_
↳ UINT16,
 (uint32)(amp((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)(amp((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]

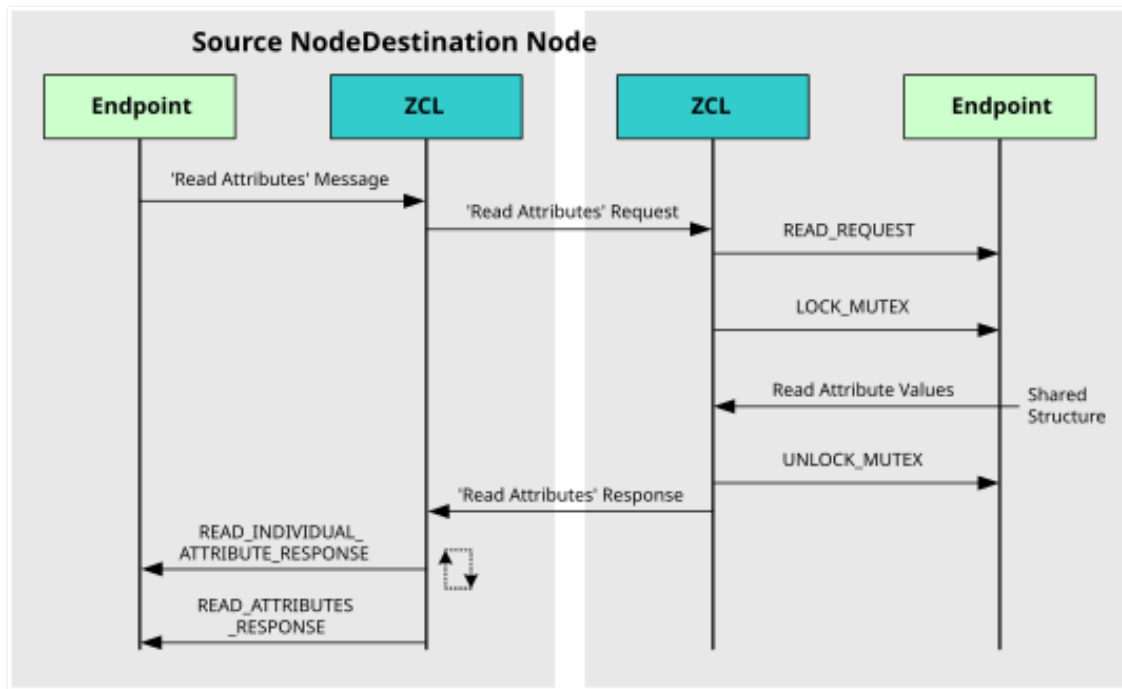
**2. On Destination Node** On receiving the ‘read attributes’ request, the ZCL software on the destination node performs the following steps:

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_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).
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.

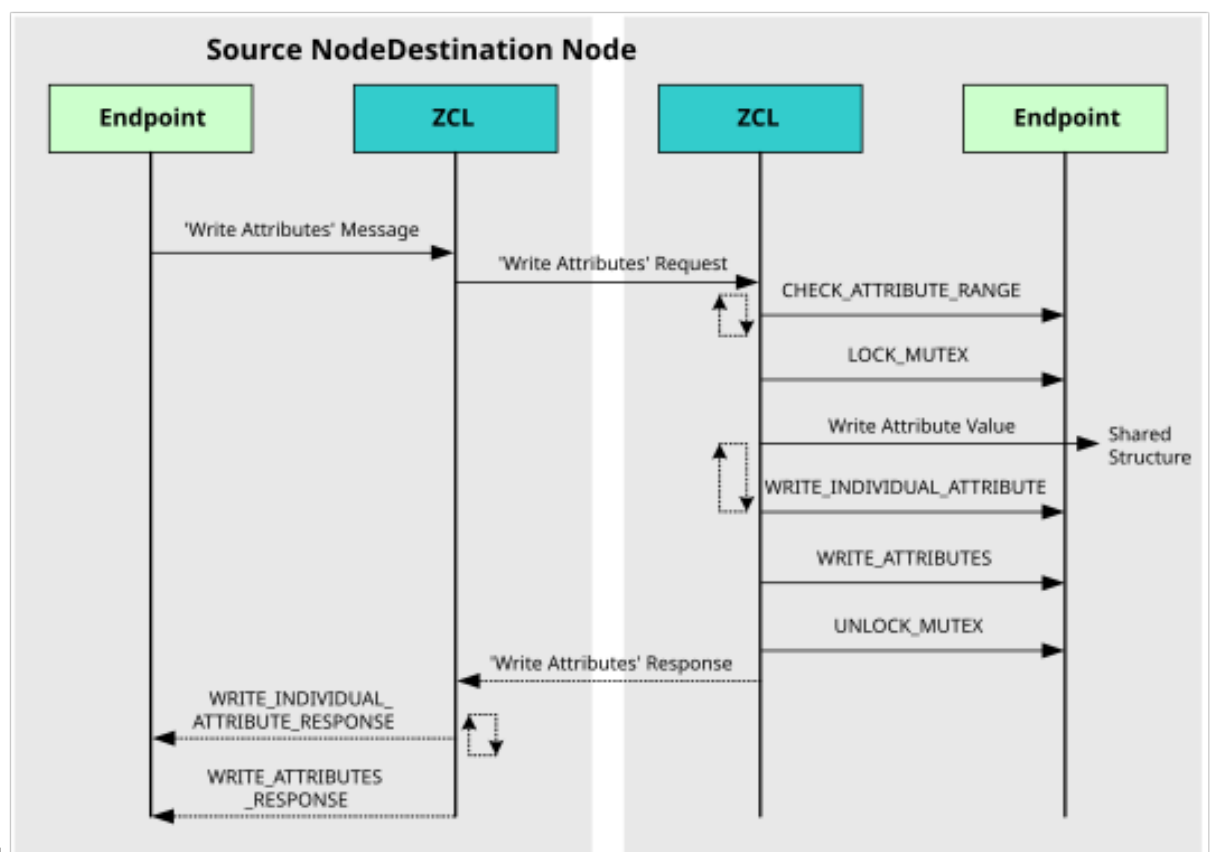
**Note:** If an ‘undivided write attributes’ request is received, an individual failed write would render the whole update process unsuccessful.

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\_RESPONSE** 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 `pZP-Sevent` 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 Section 2.3.2). 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_RESPONSES |                                    |
| 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_t 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 Event)                              | Status (in 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 <b>zcl_options.h</b> 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<br>MENT_END_POINT_NO' | None                                    | Key Establishment cluster has not been registered correctly.                                                                                                           |
| E_ZCL_ERR_KEY_ESTAB<br>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_tStatus 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 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.

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
- *psDestinationAddressPointer* 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_tStatus 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
- *bIsClientAttribute*: 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 over-ridden
- *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_t u16ClusterEnum;
 bool_t bIsManufacturerSpecificCluster;
 uint8_t u8ClusterControlFlags;
 uint16_t u16NumberOfAttributes;
 tsZCL_AttributeDefinition *psAttributeDefinition;
 tsZCL_SceneExtensionTable *psSceneExtensionTable;
#ifdef ZCL_COMMAND_DISCOVERY_SUPPORTED
 uint8_t 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_ZCLSendSecurity enumerations (see Section 7.1.6): E_ZCL_SECURITY_NETWORK, E_ZCL_SECURITY_APPLINK, E_ZCL_SECURITY_TEMP_APPLINK. 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_t u16AttributeEnum;
 uint8_t u8AttributeFlags;
 teZCL_ZCLAttributeType eAttributeDataType;
 uint16_t u16OffsetFromStructBase;
 uint16_t 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.
- `u16OffsetFromStructBase` 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 {
 uint16_t u16GroupAddress;
 uint16_t u16DestinationAddress;
 uint64_t 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.
- u16MinimumReportingInterval 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_t 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_t u8CommandId;
 uint8_t 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_t 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     | <code>E_ZCL_CF_RX</code> | Command is generated by the client and received by the server |
| 1     | <code>E_ZCL_CF_TX</code> | Command is generated by the server and received by the client |
| 2     | -                        | Reserved                                                      |
| 3     | <code>E_ZCL_CF_MS</code> | 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`C`ommandsReceivedDiscoveryIndividualResponse** 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.
  - **sAttributeDiscoveryExtendedResponse** 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_BLOCKING        | Perform a non-blocking bound transmission, with acknowledgments                 |
| E_ZCL_AM_BOUND_NON_BLOCKING_NO_ACK | 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   | All End Devices                                                                         |
| ZPS_E_APL_AF_BROADCAST_RX_ON | Nodes on which the radio receiver remains enabled when the node is idle (e.g. sleeping) |
| ZPS_E_APL_AF_BROADCAST_ZC_ZR | 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 <b>zcl_options.h</b> file                |
| E_ZCL_CMDS_UNSUP_MANUF_CLI     | Manufacturer-specific cluster command is not supported or has unknown manufacturer code |
| E_ZCL_CMDS_UNSUP_MANUF_GE      | Manufacturer-specific ZCL command is not supported 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_SPACE  | 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_ATT    | Periodic reports cannot be produced for this attribute                                  |
| E_ZCL_CMDS_INVALID_DATA_TYPE   | 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_STATE  | Setting the specified values would put device into an inconsistent state on start-up    |
| E_ZCL_CMDS_DEFINED_OUT_OF_BAND | Attempt has been made to write to attribute using an out-of-band method or not over-air |
| E_ZCL_CMDS_HARDWARE_FAILURE    | Command was unsuccessful due to hardware failure                                        |
| E_ZCL_CMDS_SOFTWARE_FAILURE    | Command was unsuccessful due to software failure                                        |
| E_ZCL_CMDS_CALIBRATION_ERROR   | Error occurred during calibration                                                       |
| E_ZCL_CMDS_UNSUPPORTED_CLUSTER | 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_MISMATCH   | Indicates that source endpoint does not match endpoint in mirror |
| E_ZCL_ATTR_REPORT_ADDR_MISMATCH | 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 writ       |
| 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 unsup    |
| 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 client    |
| 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 sp     |
| 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 for     |
| 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 synchronizer      |
| 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 cor   |
| 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 found |
| 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 Enumeration | Type | 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 re-  
ported| |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 com-  
mands (that can be generated) included in a ‘command discovery’ response have been re-  
ported| |E\_ZCL\_CBET\_DISCOVER\_INDIVIDUAL\_ATTRIBUTE\_EXTENDED\_RESPONSE|Generated  
for each attribute included in a ‘discover attributes extended’ re-  
sponse| |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 fu-  
ture use| |E\_ZCL\_CBET\_REPORT\_INDIVIDUAL\_ATTRIBUTE|Reserved for fu-  
ture 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 mod-  
ule 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
 uint8_t u8ZCLVersion;
#ifdef CLD_BAS_ATTR_APPLICATION_VERSION
 uint8_t u8ApplicationVersion;
#endif
#ifdef CLD_BAS_ATTR_STACK_VERSION
 uint8_t u8StackVersion;
#endif
#ifdef CLD_BAS_ATTR_HARDWARE_VERSION
 uint8_t u8HardwareVersion;
#endif
#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
 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 Section 6.1.14) 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 `au8DateCode` and the field `sDataCode.u8MaxLength` is 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: `u8ZCLVersion`, `ePowerSource` 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.sBasicCluster.u16ClusterRevision = CLD_BAS_CLUSTER_REVISION;
```

On a battery-powered Dimmer Switch:

```
sSwitch.sLocalBasicCluster.u8ZCLVersion = 0x01;
sSwitch.sLocalBasicCluster.ePowerSource = E_CLD_BAS_PS_BATTERY;
sSwitch.sBasicCluster.u16ClusterRevision = 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_BATTE  | 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
 uint16_t u16MainsVoltage;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_FREQUENCY
 uint8_t u8MainsFrequency;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_ALARM_MASK
 uint8_t u8MainsAlarmMask;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_VOLTAGE_MIN_THRESHOLD
 uint16_t u16MainsVoltageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_VOLTAGE_MAX_THRESHOLD
 uint16_t u16MainsVoltageMaxThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_VOLTAGE_DWELL_TRIP_POINT
 uint16_t u16MainsVoltageDwellTripPoint;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_VOLTAGE
 uint8_t u8BatteryVoltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_PERCENTAGE_REMAINING
 uint8_t u8BatteryPercentageRemaining;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_MANUFACTURER
 tsZCL_CharacterString sBatteryManufacturer;
 uint8_t au8BatteryManufacturer[16];
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_SIZE
 uint8_t u8BatterySize;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_AHR_RATING
```

(continues on next page)

(continued from previous page)

```

 uint16_t u16BatteryAHRating;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_QUANTITY
 uint8_t u8BatteryQuantity;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_RATED_VOLTAGE
 uint8_t u8BatteryRatedVoltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_ALARM_MASK
 zbm8_t u8BatteryAlarmMask;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_VOLTAGE_MIN_THRESHOLD
 uint8_t u8BatteryVoltageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD1
 uint8_t u8BatteryVoltageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD2
 uint8_t u8BatteryVoltageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD3
 uint8_t u8BatteryVoltageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_MIN_THRESHOLD
 uint8_t u8BatteryPercentageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD1
 uint8_t u8BatteryPercentageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD2
 uint8_t u8BatteryPercentageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD3
 uint8_t u8BatteryPercentageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_ALARM_STATE
 zbm32_t u32BatteryAlarmState;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_VOLTAGE
 uint8_t u8Battery2Voltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_PERCENTAGE_REMAINING
 uint8_t u8Battery2PercentageRemaining;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_MANUFACTURER
 tsZCL_CharacterString_t sBattery2Manufacturer;
 uint8_t au8Battery2Manufacturer[16];
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_SIZE
 zenum8_t u8Battery2Size;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_AHR_RATING
 uint16_t u16Battery2AHRating;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_QUANTITY
 uint8_t u8Battery2Quantity;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_RATED_VOLTAGE
 uint8_t 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
 uint8 u8Battery2VoltageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD1
 uint8 u8Battery2VoltageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD2
 uint8 u8Battery2VoltageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD3
 uint8 u8Battery2VoltageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_MIN_THRESHOLD
 uint8 u8Battery2PercentageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD1
 uint8 u8Battery2PercentageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD2
 uint8 u8Battery2PercentageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD3
 uint8 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
 uint16 u16Battery3AHRating;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_QUANTITY
 uint8 u8Battery3Quantity;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_RATED_VOLTAGE
 uint8 u8Battery3RatedVoltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_ALARM_MASK
 zbmap8 u8Battery3AlarmMask;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_VOLTAGE_MIN_THRESHOLD
 uint8 u8Battery3VoltageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD1
 uint8 u8Battery3VoltageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD2

```

(continues on next page)

(continued from previous page)

```

 uint8_t u8Battery3VoltageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD3
 uint8_t u8Battery3VoltageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_MIN_THRESHOLD
 uint8_t u8Battery3PercentageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD1
 uint8_t u8Battery3PercentageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD2
 uint8_t u8Battery3PercentageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD3
 uint8_t u8Battery3PercentageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_ALARM_STATE
 zbm32_t u32Battery3AlarmState;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
 uint8_t u8AttributeReportingStatus;
#endif
#endif
uint16_t 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 pre-defined threshold - see below) |
| 1   | Over-voltage alarm (triggered when measured RMS mains voltage rises above a pre-defined 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  | - Minimum battery voltage threshold, in units of 100 mV                                                   |
| 0x39  |                                                                                                           |
| 0x3A  | Mains power supply has been lost or is unavailable - that is, the device is now running on battery power. |
| 0x3B  | - Reserved                                                                                                |
| 0xFF  |                                                                                                           |

- `u8BatteryVoltageThreshold1` 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 `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.
- `u8BatteryVoltageThreshold3` 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 `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.
- `u8BatteryPercentageThreshold2` 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 `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                                                     |
|-------------------------|-----------------------------------------------------------------|
| <b>Bits for Battery</b> |                                                                 |
| 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 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:** [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
 uint16_t u16IdentifyTime;
#ifdef CLD_IDENTIFY_ATTR_COMMISSION_STATE
 zbmap8 u8CommissionState;
#endif
#endif
 uint16_t 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_GetLastError()`.

**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
{
 uint16_t 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
{
 uint16_t 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_t 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 <code>u8CommissionState</code> 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
 uint8 u8SceneCount;
 uint8 u8CurrentScene;
 uint16 u16CurrentGroup;
 zbool bSceneValid;
 uint8 u8NameSupport;
#ifdef CLD_SCENES_ATTR_LAST_CONFIGURED_BY
 zieeeaddress u64LastConfiguredBy;
#endif
#endif
 uint16 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_ScenesCommandCopySceneRequestSend()` 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_t u16GroupId;
 uint8_t 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_t u8Mode;
 uint16_t u16FromGroupId;
 uint8_t u8FromSceneId;
 uint16_t u16ToGroupId;
 uint8_t 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 <b>bOnOff</b> to FALSE |
| 0x01          | Put the light in the on state - set <b>bOnOff</b> 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 `u16OnConfigurableDuration` and `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
{
 uint8_t u8OnOff;
 uint16_t u16OnTime;
 uint16_t 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 0xFFFFE.
- u16OffTime is the ‘off waiting time’, expressed in tenths of a second in the range 0x0000 to 0xFFFFE

**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 Section 2.4.

**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                                                                                                                                                                                        |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>E_CLD_OOSC_TY</code> | 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).          |
| <code>E_CLD_OOSC_TY</code> | 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). |
| <code>E_CLD_OOSC_TY</code> | 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
 uint8_t u8CurrentLevel;
#ifdef CLD_LEVELCONTROL_ATTR_REMAINING_TIME
 uint16_t u16RemainingTime;
#endif
 uint8_t u8Options;
#ifdef CLD_LEVELCONTROL_ATTR_ON_OFF_TRANSITION_TIME
 uint16_t u16OnOffTransitionTime;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_ON_LEVEL
 uint8_t u8OnLevel;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_ON_TRANSITION_TIME
 uint16_t u16OnTransitionTime;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_OFF_TRANSITION_TIME
 uint16_t u16OffTransitionTime;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_DEFAULT_MOVE_RATE
 uint8_t u8DefaultMoveRate;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_STARTUP_CURRENT_LEVEL
 uint8_t u8StartUpCurrentLevel;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_ATTRIBUTE_REPORTING_STATUS
 uint8_t u8AttributeReportingStatus;
#endif
#endif
 uint16_t 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    | Change | 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 Section 16.8.3.1).

**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 `u8DefaultMoveRate` 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
 uint16_t u16AlarmCount;
#endif
#endif
 uint16_t u16ClusterRevision;
} tsCLD_Alarms;
```

where:

- `u16AlarmCount` 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 on page 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.

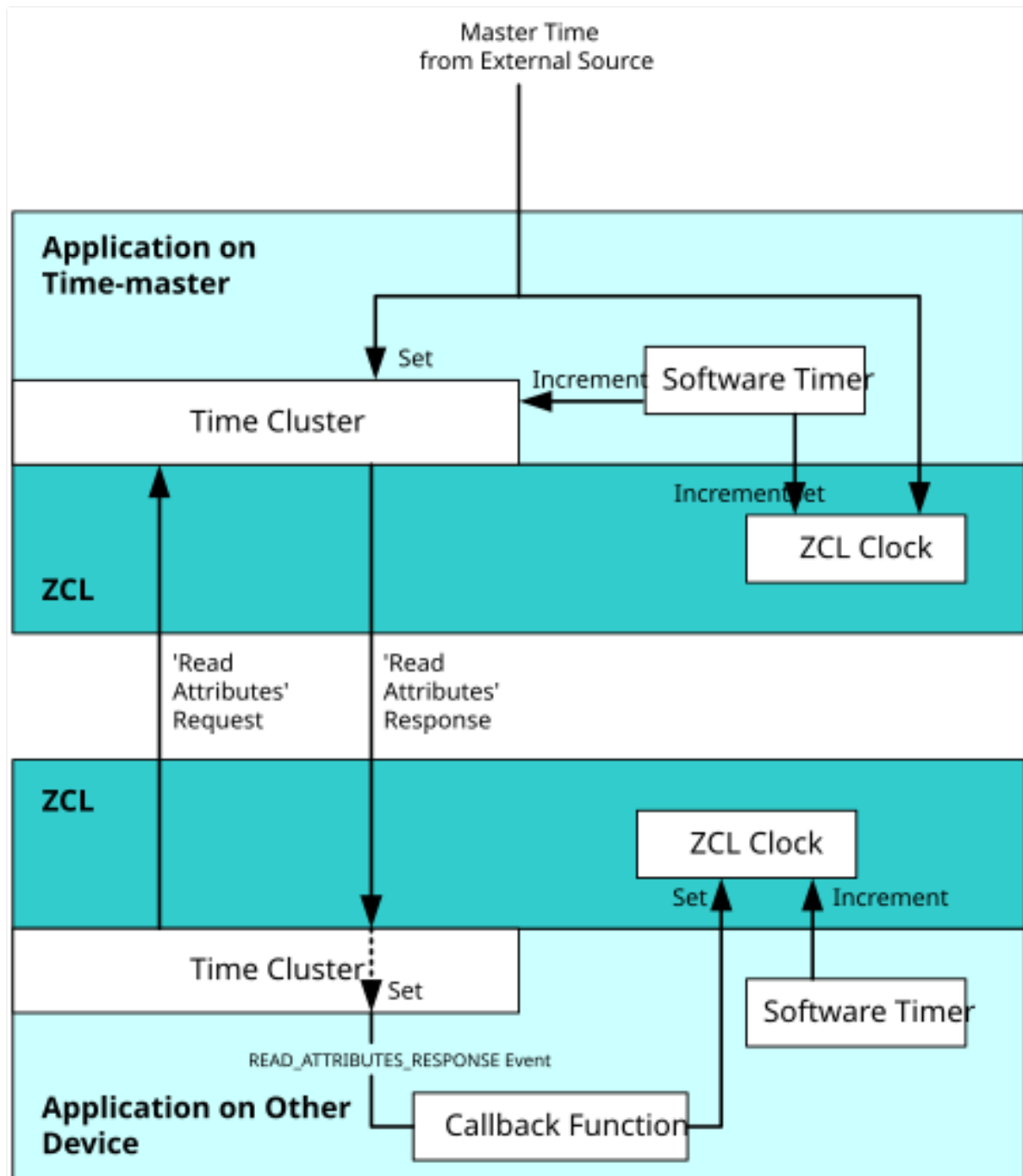
CAUTION:

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;
 uint8 au8Description[16];
#endif
#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
 uint32 u32ApplicationType;
#endif
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
 zenum8 u8AttributeReportingStatus;
#endif
 uint16 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 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
- 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

(fPresentValue 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 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 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 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 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
    - 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 <code>u8Reliability</code> is used and does not have a value of <code>NO_FAULT_DETECTED</code> |

- 0: Otherwise

| 2 | Overridden | - 1: Cluster has been over-ridden by a local mechanism

(`fPresentValue` 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 Name | 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 | Group      | 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];
 #endif
 #endif
}
```

(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
#endif
zbool bOutOfService;
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_POLARITY
 zenum8 u8Polarity;
#endif
#endif
zbool bPresentValue;
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_RELIABILITY
 zenum8 u8Reliability;
#endif
#endif
zmap8 u8StatusFlags;
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_APPLICATION_TYPE
 zuint32 u32ApplicationType;
#endif
#endif
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
 zenum8 u8AttributeReportingStatus;
#endif
#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 Section 6.1.14) 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 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
- 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 Section 6.1.14) 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 <code>u8Reliability</code> is used and does not have a value of <code>NO_FAULT_DETECTED</code> |

- 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 Cluster ID). For the Binary Input (Basic) cluster, this is 0x03.                                                                          |

`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 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
 uint32 u32MinimumOffTime;
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_MINIMUM_ON_TIME
 uint32 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
 uint32 u32ApplicationType;
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
 zenum8 u8AttributeReportingStatus;
#endif
#endif
 uint16 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 Section 6.1.14) 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 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
- 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 Section 6.1.14) 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 Cluster 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
 uint16 u16NumberOfStates;
 zbool bOutOfService;
 uint16 u16PresentValue;
#ifdef CLD_MULTISTATE_INPUT_BASIC_ATTR_RELIABILITY
 zenum8 u8Reliability;
#endif
 zbmap8 u8StatusFlags;
#ifdef CLD_MULTISTATE_INPUT_BASIC_ATTR_APPLICATION_TYPE
 uint32 u32ApplicationType;
#endif
#ifdef CLD_MULTISTATE_INPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
 zenum8 u8AttributeReportingStatus;
#endif
#endif
 uint16 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 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 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 <code>u8Reliability</code> is used and does not have a value of <code>NO_FAULT_DETECTED</code> |

- 0: Otherwise

| |2| Overridden | - 1: Cluster has been over-ridden by a local mechanism

(`fPresentValue` 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 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

**tsCLD\_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
#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 <code>u8Reliability</code> is used and does not have a value of <code>NO_FAULT_DETECTED</code> |

- 0: Otherwise

| 2| Overridden|- 1: Cluster has been over-ridden by a local mechanism

(`u16PresentValue` 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
 uint32_t u32CheckinInterval;
 uint32_t u32LongPollInterval;
 uint16_t u16ShortPollInterval;
 uint16_t u16FastPollTimeout;
#ifdef CLD_POLL_CONTROL_ATTR_CHECKIN_INTERVAL_MIN
 uint32_t u32CheckinIntervalMin;
#endif
#ifdef CLD_POLL_CONTROL_ATTR_LONG_POLL_INTERVAL_MIN
 uint32_t u32LongPollIntervalMin;
#endif
#ifdef CLD_POLL_CONTROL_ATTR_FAST_POLL_TIMEOUT_MAX
 uint16_t u16FastPollTimeoutMax;
#endif
#endif
 uint16_t 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`  $\geq$  `u32LongPollInterval`  $\geq$  `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`.

| <b>u8CommandId</b>                 | <b>Enumeration</b> | <b>Description/Payload Type</b>                                                                    |
|------------------------------------|--------------------|----------------------------------------------------------------------------------------------------|
| <b>On Client</b>                   |                    |                                                                                                    |
| <code>E_CLD_POLL_CONTROL_CM</code> |                    | A Check-in command has been received by the client.                                                |
| <b>On Server</b>                   |                    |                                                                                                    |
| <code>E_CLD_POLL_CONTROL_CM</code> |                    | 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
{
 uint32_t 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
{
 uint16_t 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
 uint8_t u8TotalProfileNum;
 bool bMultipleScheduling;
 uint8_t u8EnergyFormatting;
 bool bEnergyRemote;
 uint8_t u8ScheduleMode;
#ifdef CLD_PP_ATTR_ATTRIBUTE_REPORTING_STATUS
 uint8_t u8AttributeReportingStatus;
#endif
#endif
 uint16_t 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.

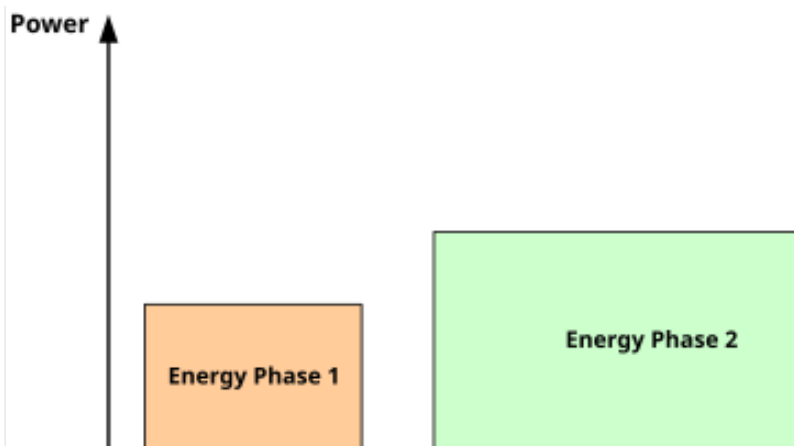
**Parent topic:** [Power Profile Cluster](#)

**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_PP_CallbackMessage`, 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 attribute `bEnergyRemote` 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_PP_PowerProfileScheduleConstraintsNotificationSend()`. The client application can alternatively request these schedule constraints from the server by calling `eCLD_PP_PowerProfileScheduleConstraintsReqSend()`.

**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_PP_EnergyPhasesScheduleNotificationSend()` 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_NOTIFICATION` 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_PP_EnergyPhasesScheduleStateNotificationSend()` 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 Section 21.5.4), 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_PP_CallBackMessage` 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_PP_CallBackMessage;
```

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_PP_CallBackMessage` 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_POWER_PROFILE_REQUEST                    |             | The server (appliance) receives a Power Profile Request.<br><b>tsCLD_PP_PowerProfileReqPayload</b>                                                                                                   |
| E_CLD_PP_CMD_POWER_PROFILE_STATE_REQUEST              |             | The server (appliance) receives a Power Profile State Request.                                                                                                                                       |
| E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_REQUEST          |             | The server (appliance) receives a Get Power Profile Price Response, following a previously sent Get Power Profile Price Request.<br><b>tsCLD_PP_GetPowerProfilePriceRspPayload</b>                   |
| E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_REQUEST |             | The server (appliance) receives a Get Power Profile Price Extended Response, following a previously sent Get Power Profile Price Extended Request.<br><b>tsCLD_PP_GetPowerProfilePriceRspPayload</b> |

| 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_NOTIFICATION | 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 Profile 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_PPEnter *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_PPEnter` 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_PPEnter` 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_PPEnt *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\_PPEnergyPhasesScheduleStateNotificationSend**

```
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_NOTIFICATION` 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\_PP\_EnergyPhasesScheduleReqSend

```
teZCL_Status eCLD_PP_EnergyPhasesScheduleReqSend(
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 Section 21.5.4.1.

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 Section 21.10.9). 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 Section 21.10.9). 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 Section 21.10.10). 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\_PP\_EnergyPhasesScheduleNotificationSend

```
teZCL_Status eCLD_PP_EnergyPhasesScheduleNotificationSend(
 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\_NOTIFICATION 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\_PP\_EnergyPhasesScheduleStateReqSend

```
teZCL_Status eCLD_PP_EnergyPhasesScheduleStateReqSend(
 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 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 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.

| Enumera-<br>tion | 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_PP_CallbackMessage;

```

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)
  - `psGetOverallSchedulePriceRspPayload` is a pointer to the payload of a Get Overall Schedule Price Response (see Section 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
{
 uint8_t u8PowerProfileId;
 uint8_t u8NumOfTransferredEnergyPhases;
 uint8_t u8NumOfScheduledEnergyPhases;
 uint8_t u8ActiveEnergyPhaseId;
 tsCLD_PP_EnergyPhaseDelay
 asEnergyPhaseDelay[CLD_PP_NUM_OF_ENERGY_PHASES];
 tsCLD_PP_EnergyPhaseInfo
 asEnergyPhaseInfo[CLD_PP_NUM_OF_ENERGY_PHASES];
 bool bPowerProfileRemoteControl;
 uint8_t u8PowerProfileState;
 uint16_t u16StartAfter;
 uint16_t 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
{
 uint8_t 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
{
 uint8_t u8PowerProfileId;
 uint8_t 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
{
 uint8_t u8PowerProfileId;
 uint16_t u16StartAfter;
 uint16_t 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
{
 uint8_t u8Options;
 uint8_t u8PowerProfileId;
 uint16_t 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
{
 uint8_t u8PowerProfileId;
 uint16_t u16Currency;
 uint32_t u32Price;
 uint8_t 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
{
 uint16_t u16Currency;
 uint32_t u32Price;
 uint8_t 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
{
 uint8_t u8EnergyPhaseId;
 uint8_t u8MacroPhaseId;
 uint16_t u16ExpectedDuration;
 uint16_t u16PeakPower;
 uint16_t u16Energy;
 uint16_t 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 ( $\square$  `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
{
 uint8_t u8EnergyPhaseId;
 uint16_t 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\_PowerProfileRecord** This structure contains information about the current state of a power profile.

```
typedef struct
{
 uint8_t u8PowerProfileId;
 uint8_t u8EnergyPhaseId;
 bool bPowerProfileRemoteControl;
 enum8_t 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 u16APSFCEFailure;
#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 u16AverageMACRetryPerAPSMessagesSent;
#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
u16APSFCEFailure is reserved for future use
u16APSunauthorizedKey is reserved for future use
u16NWKDecryptFailure is reserved for future use
u16APSDDecryptFailure 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
```

- `u16AverageMACRetryPerAPSMessagesSent` 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 Section 22.2). 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
 uint16_t u16MeasuredValue;
 uint16_t u16MinMeasuredValue;
 uint16_t u16MaxMeasuredValue;
#ifdef CLD_ILLMEAS_ATTR_TOLERANCE
 uint16_t u16Tolerance;
#endif
#ifdef CLD_ILLMEAS_ATTR_LIGHT_SENSOR_TYPE
 uint8_t eLightSensorType;
#endif
#ifdef CLD_ILLMEAS_ATTR_ATTRIBUTE_REPORTING_STATUS
 uint8_t u8AttributeReportingStatus;
#endif
#endif
 uint16_t u16ClusterRevision;
} tsCLD_IlluminanceMeasurement;
```

where:

- **u16MeasuredValue** is a mandatory attribute representing the measured illuminance in logarithmic form, calculated as  $(10000 \times \log_{10} \text{Illuminance}) + 1$ , where the illuminance is measured in Lux (lx). The possible illumination values are in the range 1 lx to  $3.576 \times 10^6$  lx, corresponding to attribute values of 1 to 0xFFFF. 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.
- *blsServer*: 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_TemperatureMeasurement` which 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
 uint16 u16Tolerance;
 #endif
#ifdef CLD_FLOWMEAS_ATTR_ATTRIBUTE_REPORTING_STATUS
 enum8 u8AttributeReportingStatus;
 #endif
 #endif
 uint16 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 (m<sup>3</sup>/h). The range of possible values is 0x0000 (0 m<sup>3</sup>/h) to 0xFFFF (representing 6553.4 m<sup>3</sup>/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 Section 7.2.

**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:

- `u16MeasuredValue` is a mandatory attribute representing the measured relatively humidity as a percentage in steps of 0.01%, as follows:
  - `u16MeasuredValue` = 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 0xFFFF 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 occupancy 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
 uint16 u16PIROccupiedToUnoccupiedDelay;
#endif
#ifdef CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_DELAY
 uint8 u8PIRUnoccupiedToOccupiedDelay;
#endif
#ifdef CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
 uint8 u8PIRUnoccupiedToOccupiedThreshold;
#endif
#ifdef CLD_OS_ATTR_ULTRASONIC_OCCUPIED_TO_UNOCCUPIED_DELAY
 uint16 u16UltrasonicOccupiedToUnoccupiedDelay;
#endif
#ifdef CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_DELAY
 uint8 u8UltrasonicUnoccupiedToOccupiedDelay;
#endif
#ifdef CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
 uint8 u8UltrasonicUnoccupiedToOccupiedThreshold;
#endif
#ifdef CLD_OS_ATTR_ATTRIBUTE_REPORTING_STATUS
 zenum8 u8AttributeReportingStatus;
#endif
#endif
 uint16 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

- `u16UltrasonicOccupiedToUnoccupiedDelay` 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 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:** *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
 uint16 u16ACFrequency;
 #endif
 #ifdef CLD_ELECTMEAS_ATTR_RMS_VOLTAGE
 uint16 u16RMSVoltage;
 #endif
 #ifdef CLD_ELECTMEAS_ATTR_RMS_CURRENT
 uint16 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
 uint16 u16ApparentPower;
 #endif
 #ifdef CLD_ELECTMEAS_ATTR_POWER_FACTOR
 zint8 i8PowerFactor;
 #endif
 #ifdef CLD_ELECTMEAS_ATTR_AC_VOLTAGE_MULTIPLIER
 uint16 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 device 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 attribute is also described in Section 2.4.

**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 Section 31.9.

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
 uint8_t u8CurrentHue;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_CURRENT_SATURATION
 uint8_t u8CurrentSaturation;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_REMAINING_TIME
 uint16_t u16RemainingTime;
#endif
 uint16_t u16CurrentX;
 uint16_t u16CurrentY;
#ifdef CLD_COLOURCONTROL_ATTR_DRIFT_COMPENSATION
 zenum8 u8DriftCompensation;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COMPENSATION_TEXT
 tsZCL_CharacterString sCompensationText;
 uint8_t au8CompensationText[
 CLD_COLOURCONTROL_COMPENSATION_TEXT_MAX_STRING_LENGTH];
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIREL
 uint16_t u16ColourTemperatureMired;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_MODE
 zenum8 u8ColourMode;
#endif
 zbitmap8 u8Options;
 /* Defined Primaries Information attribute set */
```

(continues on next page)

(continued from previous page)

```

#ifndef CLD_COLOURCONTROL_ATTR_NUMBER_OF_PRIMARIES
 uint8 u8NumberOfPrimaries;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_1_X
 uint16 u16Primary1X;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_1_Y
 uint16 u16Primary1Y;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_1_INTENSITY
 uint8 u8Primary1Intensity;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_2_X
 uint16 u16Primary2X;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_2_Y
 uint16 u16Primary2Y;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_2_INTENSITY
 uint8 u8Primary2Intensity;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_3_X
 uint16 u16Primary3X;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_3_Y
 uint16 u16Primary3Y;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_3_INTENSITY
 uint8 u8Primary3Intensity;
#endif
/* Additional Defined Primaries Information attribute set */
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_4_X
 uint16 u16Primary4X;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_4_Y
 uint16 u16Primary4Y;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_4_INTENSITY
 uint8 u8Primary4Intensity;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_5_X
 uint16 u16Primary5X;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_5_Y
 uint16 u16Primary5Y;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_5_INTENSITY
 uint8 u8Primary5Intensity;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_6_X
 uint16 u16Primary6X;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_6_Y
 uint16 u16Primary6Y;
#endif
#ifndef CLD_COLOURCONTROL_ATTR_PRIMARY_6_INTENSITY
 uint8 u8Primary6Intensity;
#endif
/* Defined Colour Points Settings attribute set */
#ifndef CLD_COLOURCONTROL_ATTR_WHITE_POINT_X
 uint16 u16WhitePointX;

```

(continues on next page)

(continued from previous page)

```

#endif
#ifdef CLD_COLOURCONTROL_ATTR_WHITE_POINT_Y
 uint16_t u16WhitePointY;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_X
 uint16_t u16ColourPointRX;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_Y
 uint16_t u16ColourPointRY;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_INTENSITY
 uint8_t u8ColourPointRIntensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_X
 uint16_t u16ColourPointGX;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_Y
 uint16_t u16ColourPointGY;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_INTENSITY
 uint8_t u8ColourPointGIntensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_X
 uint16_t u16ColourPointBX;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_Y
 uint16_t u16ColourPointBY;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_INTENSITY
 uint8_t u8ColourPointBIntensity;
#endif
/* Colour information attribute set */
#ifdef CLD_COLOURCONTROL_ATTR_ENHANCED_CURRENT_HUE
 uint16_t u16EnhancedCurrentHue;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_ENHANCED_COLOUR_MODE
 uint8_t u8EnhancedColourMode;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_ACTIVE
 uint8_t u8ColourLoopActive;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_DIRECTION
 uint8_t u8ColourLoopDirection;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_TIME
 uint16_t u16ColourLoopTime;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_START_ENHANCED_HUE
 uint16_t u16ColourLoopStartEnhancedHue;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_STORED_ENHANCED_HUE
 uint16_t u16ColourLoopStoredEnhancedHue;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_CAPABILITIES
 uint16_t u16ColourCapabilities;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIREDD_PHY_MIN
 uint16_t u16ColourTemperatureMiredPhyMin;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIREDD_PHY_MAX
 uint16_t u16ColourTemperatureMiredPhyMax;

```

(continues on next page)

(continued from previous page)

```

#endif
#ifdef CLD_COLOURCONTROL_ATTR_COUPLE_COLOUR_TEMPERATURE_TO_LEVEL_MIN_
↪ MIREd
 uint16_t u16CoupleColourTempToLevelMinMired;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_STARTUP_COLOUR_TEMPERATURE_MIREd
 uint16_t u16StartupColourTemperatureMired;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_ATTRIBUTE_REPORTING_STATUS
 uint8_t u8AttributeReportingStatus;
#endif
#endif
uint16_t 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:  

$$\text{hue} = \text{u8CurrentHue} \times 360/254.$$
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:  $\text{saturation} = \text{u8CurrentSaturation}/254$ . 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:  

$$x = \text{u16CurrentX}/65536.$$
- `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:  

$$y = \text{u16CurrentY}/65536.$$
- `u8DriftCompensation` indicates the mechanism, if any, is being used to compensate for colour/intensity drift over time. One of the following values is specified:

| <code>u8DriftCompensation</code> | 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:  
$$T = 1000000 / u16ColourTemperatureMired$$
. This attribute is only valid when the attribute `u8ColourMode` 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:

| <code>u8ColourMode</code> | 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 (`u16ColourTemperatureMired`) | 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 <code>bOnOff</code> 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*’ *AttributeSet*, 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 on page 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:

| <b>u8ColourMode</b> | <b>Colour Control Method/Attributes</b> |
|---------------------|-----------------------------------------|
| 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 ``u16CoupleColourTempToLevelMinMired`` 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 [Section 2.4](../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:

$$\text{Saturation} = \text{u8CurrentSaturation}/254$$

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:

$T = 1000000 / u16ColourTemperature \cdot \text{Mired}$

**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  $H_i$  (where  $i$  is the relevant table index) and the linear interpolation value is  $interp$ , the 'enhanced' hue is given by the formula:

$$\text{Enhanced hue} = H_i + (interp/255) \times (H_{i+1} - H_i)$$

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 Section 31.5.7.

**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 or 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_MIREDD_PHY_MIN,
E_CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIREDD_PHY_MAX,
E_CLD_COLOURCONTROL_ATTR_COUPLE_COLOUR_TEMPERATURE_TO_LEVEL_MIN_
↪MIREDD,
E_CLD_COLOURCONTROL_ATTR_STARTUP_COLOUR_TEMPERATURE_MIREDD = 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 ‘ColourCapabilities’ 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    |
|------------------------|-------------|----------------|
| <b>Mandatory in...</b> | Thermostat  |                |
| <b>Optional in...</b>  |             | 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 Section 33.10.

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 327.67°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)
- 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:
  - $i8LocalTemperatureCalibration = 100 \times \text{offset in degrees Celsius}$
  - 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 offsets 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).

- `i16UnoccupiedHeatingSetpoint` 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 `i16MinHeatSetpointLimit` and `i16MaxHeatSetpointLimit`. If it is not known whether the space is occupied, this attribute will not be used (`i16OccupiedHeatingSetpoint` will be used instead).

**Note:** `i16UnoccupiedCoolingSetpoint` must always be greater in value than `i16UnoccupiedHeatingSetpoint` 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 <code>eSystemMode</code> )       |
|-------|---------------------|---------------------------------------------|
| 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   | <b>Initialisation failure</b> (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 Section 33.9.3)

### 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:

- `u8CommandId` indicates the type of Thermostat cluster command that has been received - there is only one possibility: `E_CLD_THERMOSTAT_CMD_SETPOINT_RAISE_LOWER`
- `uMessage` 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 structure 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 Section 34.5.2, 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:

- | • <b>Function</b>       | <b>Page</b> |
|-------------------------|-------------|
| • 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 Section 34.2).

```
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        |
|------------------------|-------------|--------------------|
| <b>Mandatory in...</b> |             |                    |
| <b>Optional in...</b>  | 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:

- | • <b>Function</b>                                              | <b>Page</b> |
|----------------------------------------------------------------|-------------|
| • <code>eCLD_ThermostatUIConfigCreateThermostatUIConfig</code> | 754         |
| • <code>eCLD_ThermostatUIConfigConvertTemp</code>              | 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_MODE_CELSIUS    | Display temperature in Celsius    |
| E_CLD_THERMOSTAT_UI_CONFIG_TEMPERATURE_DISPLAY_MODE_FAHRENHEIT | 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          |
|------------------------|-------------|----------------------|
| <b>Mandatory in...</b> | Door Lock   | Door Lock Controller |
| <b>Optional in...</b>  |             | 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
#ifdef
 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 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:** [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.

| <b>u8CommandId Enumeration</b>          | <b>Description</b>                                                |
|-----------------------------------------|-------------------------------------------------------------------|
| <code>E_CLD_DOOR_LOCK_CMD_LOCK</code>   | A lock request command has been received by the cluster server    |
| <code>E_CLD_DOOR_LOCK_CMD_UNLOCK</code> | 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:

- | • <b>Function</b>                                        | <b>Page</b> |
|----------------------------------------------------------|-------------|
| • <code>eCLD_DoorLockCreateDoorLock</code>               | 766         |
| • <code>eCLD_DoorLockSetLockState</code>                 | 768         |
| • <code>eCLD_DoorLockGetLockState</code>                 | 769         |
| • <code>eCLD_DoorLockCommandLockUnlockRequestSend</code> | 770         |
| • <code>eCLD_DoorLockSetSecurityLevel</code>             | 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 structure 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                   |
|-----------------------------------------------------------|-------|---------------------------|----------------------------|--------------------------|
| <code>E_CLD_IASZONE_TYPE_STANDARI</code>                  | 0x000 | Standard CIE              | System alarm               | -                        |
| <code>E_CLD_IASZONE_TYPE_MOTION_SENSOR</code>             | 0x000 | Motion sensor             | Intrusion indication       | Presence indication      |
| <code>E_CLD_IASZONE_TYPE_CONTACT_SWITCH</code>            | 0x001 | Contact switch            | First portal open close    | Second portal open-close |
| <code>E_CLD_IASZONE_TYPE_FIRE_SENSOR</code>               | 0x002 | Fire sensor               | Fire indication            | -                        |
| <code>E_CLD_IASZONE_TYPE_WATER_SENSOR</code>              | 0x002 | Water sensor              | Water overflow indication  | -                        |
| <code>E_CLD_IASZONE_TYPE_GAS_SENSOR</code>                | 0x002 | Gas sensor                | Carbon monoxide indication | Cooking indication       |
| <code>E_CLD_IASZONE_TYPE_PERSONAL_EMERGENCY_DEVICE</code> | 0x002 | Personal emergency device | Fall/concussion            | Emergency button         |
| <code>E_CLD_IASZONE_TYPE_VIBRATION_MOVEMENT_SENSOR</code> | 0x002 | Vibration movement sensor | Movement indication        | Vibration                |
| <code>E_CLD_IASZONE_TYPE_REMOTE_CONTROL</code>            | 0x010 | Remote control            | Panic                      | Emergency                |
| <code>E_CLD_IASZONE_TYPE_KEY_FOB</code>                   | 0x011 | Key fob                   | Panic                      | Emergency                |
| <code>E_CLD_IASZONE_TYPE_KEYPAD</code>                    | 0x021 | Keypad                    | Panic                      | Emergency                |
| <code>E_CLD_IASZONE_TYPE_STANDARD_WARNING_DEVICE</code>   | 0x022 | Standard warning device   | -                          | -                        |
| <code>E_CLD_IASZONE_TYPE_INVALID_ZONE_TYPE</code>         | 0xFF  | 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 alarmed               |
| 1     | Alarm2:1 - Opened or alarmed0 - Closed or not alarmed               |
| 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                                                                                          |
|------------------------------------------|------------------------------------------------------------------------------------------------------|
| <code>E_CLD_IASZONE_CMD_ZONE_ENI</code>  | An IAS Zone Enroll Response has been received by the cluster server                                  |
| <code>E_CLD_IASZONE_CMD_ZONE_STA</code>  | An IAS Zone Status Change Notification has been received by the cluster client                       |
| <code>E_CLD_IASZONE_CMD_ZONE_ENI</code>  | An IAS Zone Enroll Request has been received by the cluster client                                   |
| <code>E_CLD_IASZONE_CMD_INITIATE_</code> | An IAS Zone Normal Operation Mode Initiation Request command has been received by the cluster server |
| <code>E_CLD_IASZONE_CMD_INITIATE_</code> | 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 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
- *eIASZoneType*: 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;
 uint8 u8ZoneId;
 uint16 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 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
{
 uint16_t 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

| Com-mand               | 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()                                                                                                                                                                              |
| By-pass                | 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                                                                      |
|----------------------------------------|----------------------------------------------------------------------------------|
| <b>Server Events</b>                   |                                                                                  |
| 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_ID           | An IAS ACE Get Zone ID Map command has been received by the server               |
| E_CLD_IASACE_CMD_GET_ZONE_INFO         | An IAS ACE Get Zone Information command has been received by the server          |
| E_CLD_IASACE_CMD_GET_PANEL_STATUS      | An IAS ACE Get Panel Status command has been received by the server              |
| E_CLD_IASACE_CMD_GET_BYPASS_ZONE_LIST  | An IAS ACE Get Bypassed Zone List command has been received by the server        |
| E_CLD_IASACE_CMD_GET_ZONE_STATUS       | An IAS ACE Get Zone Status command has been received by the server               |
| <b>Client Events</b>                   |                                                                                  |
| E_CLD_IASACE_CMD_ARM_RESP              | An IAS ACE Arm Response command has been received by the client                  |
| E_CLD_IASACE_CMD_GET_ZONE_ID_RESP      | An IAS ACE Get Zone ID Map Response command has been received by the client      |
| E_CLD_IASACE_CMD_GET_ZONE_INFO_RESP    | An IAS ACE Get Zone Information Response command has been received by the client |
| E_CLD_IASACE_CMD_ZONE_STATUS_CHANGED   | An IAS ACE Zone Status Changed command has been received by the client           |
| E_CLD_IASACE_CMD_PANEL_STATUS_CHANGED  | An IAS ACE Panel Status Changed command has been received by the client          |
| E_CLD_IASACE_CMD_GET_PANEL_STATUS_RESP | An IAS ACE Get Panel Status Response command has been received by the client     |
| E_CLD_IASACE_CMD_SET_BYPASS_ZONE_LIST  | An IAS ACE Set Bypassed Zone List command has been received by the client        |
| E_CLD_IASACE_CMD_BYPASS_RESP           | An IAS ACE Bypass Response command has been received by the client               |
| E_CLD_IASACE_CMD_GET_ZONE_STATUS_RESP  | 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 (specified 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 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                           |

- `eZoneStatus` 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 Section 38.8.1):

| 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
{
 uint8_t u8NumOfZones;
 uint8_t *pu8ZoneID;
 tsZCL_CharacterString_t 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
{
 uint8_t 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
{
 uint8_t u8NumofZones;
 uint8_t *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
{
 uint8_t u8StartingZoneID;
 uint8_t 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 alarmed

| |1| Alarm2:

1 - Opened or alarmed

0 - Closed or not alarmed

| |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                          |
|-------------------------------|---------|----------------------------------|
| <code>au16ZoneIDMap[0]</code> | 01: 15  | 0x000x01: 0x0F                   |
| <code>au16ZoneIDMap[1]</code> | 01: 15  | 0x100x11: 0x1F                   |
| :                             | :       | :                                |
| <code>au16ZoneIDMap[N]</code> | 01:n:15 | 16N16N + 0x1:16N + 0xn:16N + 0xF |

**E\_CLD\_IASACE\_CMD\_GET\_ZONE\_INFO\_RESP Data**

```
typedef struct
{
 uint8_t u8ZoneID;
 uint16_t u16ZoneType;
 uint32_t u64IeeeAddress;
 tsZCL_CharacterString_t 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
{
 uint8_t u8ZoneID;
 uint16_t eZoneStatus;
 uint8_t eAudibleNotification;
 tsZCL_CharacterString_t 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
{
 uint8_t u8NumofZones;
 uint8_t *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
{
 bool bZoneStatusComplete;
 uint8_t u8NumofZones;
 uint8_t *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
 uint16_t u16MaxDuration;
 #endif
 uint16_t u16ClusterRevision;
} tsCLD_IASWD;
```

where:

- `u16MaxDuration` 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-<br>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<br>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                                        |
|------------------------|-------------|----------------------------------------------------|
| <b>Mandatory in...</b> | ESP         | Smart Appliance                                    |
| <b>Optional in...</b>  |             | 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_t 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_t au48BlockThreshold[CLD_P_ATTR_BLOCK_THRESHOLD_MAX_COUNT];
 #endif
 /* Block Period Set (D.4.2.2.3) */
 #ifndef CLD_P_ATTR_START_OF_BLOCK_PERIOD
 zutctime_t utctStartOfBlockPeriod;
 #endif
 #ifndef CLD_P_ATTR_BLOCK_PERIOD_DURATION
 zuint24_t u24BlockPeriodDuration;
 #endif
 #ifndef CLD_P_ATTR_THRESHOLD_MULTIPLIER
```

(continues on next page)

(continued from previous page)

```

 uint24_t u24ThresholdMultiplier;
#endif
#ifdef CLD_P_ATTR_THRESHOLD_DIVISOR
 uint24_t u24ThresholdDivisor;
#endif
/* Commodity Set Set (D.4.2.2.4) */
#ifdef CLD_P_ATTR_COMMODITY_TYPE
 enum8 e8CommodityType;
#endif
#ifdef CLD_P_ATTR_STANDING_CHARGE
 uint32_t u32StandingCharge;
#endif
#ifdef CLD_P_ATTR_CONVERSION_FACTOR
 uint32_t u32ConversionFactor;
#endif
#ifdef CLD_P_ATTR_CONVERSION_FACTOR_TRAILING_DIGIT
 uint8_t b8ConversionFactorTrailingDigit;
#endif
#ifdef CLD_P_ATTR_CALORIFIC_VALUE
 uint32_t u32CalorificValue;
#endif
#ifdef CLD_P_ATTR_CALORIFIC_VALUE_UNIT
 enum8 e8CalorificValueUnit;
#endif
#ifdef CLD_P_ATTR_CALORIFIC_VALUE_TRAILING_DIGIT
 uint8_t b8CalorificValueTrailingDigit;
#endif
/* Block Price Information Set (D.4.2.2.5) */
if (CLD_P_ATTR_NO_TIER_BLOCK_PRICES_MAX_COUNT != 0)
 uint32_t 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))
 uint32_t 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))
 uint32_t 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))
 uint32_t 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))
 uint32_t 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))
 uint32_t 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))
 uint32_t 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))
 uint32_t 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)

```

 uint32_t 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))
 uint32_t 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))
 uint32_t 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))
 uint32_t 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))
 uint32_t 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))
 uint32_t 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))
 uint32_t 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))
 uint32_t au32Tier15BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
#ifdef CLD_P_ATTR_START_OF_BILLING_PERIOD
 time_t utctStartOfBillingPeriod;
#endif
#ifdef CLD_P_ATTR_BILLING_PERIOD_DURATION
 uint32_t u24BillingPeriodDuration;
#endif
#ifdef CLD_P_CLIENT_ATTR_PRICE_INCREASE_RANDOMIZE_MINUTES
 uint8_t u8ClientIncreaseRandomize;
#endif
#ifdef CLD_P_CLIENT_ATTR_PRICE_DECREASE_RANDOMIZE_MINUTES
 uint8_t u8ClientDecreaseRandomize;
#endif
#ifdef CLD_P_CLIENT_ATTR_COMMODITY_TYPE
 enum8_t 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\_M 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 Table 74.

**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 Section 40.5.2). 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_ACTIVE` 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 *pu8RateLabel,
 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 to 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
- *uNumberOfRecordEntries* 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
- *puAttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
- *pauRateLabel* 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\_BOUNDED. 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 Section 40.11.2), 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_FOUND       | 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_FOUND       | 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 0xFFFFFFFF (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_PERIOD_OVERFLOW | Reserved for future use (for Block mode)                                                                                                                         |
| E_SE_PRICE_NO_BLOCKS             | Reserved for future use (for Block mode)                                                                                                                         |
| E_SE_PRICE_NUMBER_OF_BLOCKS      | Reserved for future use (for Block mode)                                                                                                                         |
| E_SE_BLOCK_PERIOD_OVERFLOW       | Reserved for future use (for Block mode)                                                                                                                         |
| E_SE_BLOCK_PERIOD_DUPLICATE      | Reserved for future use (for Block mode)                                                                                                                         |
| E_SE_BLOCK_PERIOD_DATA_OLD       | Reserved for future use (for Block mode)                                                                                                                         |
| E_SE_BLOCK_PERIOD_OVERLAP        | 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 CO<sub>2</sub>, 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’

- `u32Price` 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 `u8PriceTrailingDigitAndPriceTier`
- `u32GenerationPrice` (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_F/  | Generated when a new conversion factor value becomes active                                                   |
| E_SE_PRICE_CONVERSION_F/  | 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 |
|------------------------|----------------------------|-------------|
| <b>Mandatory in...</b> | ESP<br>Load Control Device | PCT         |
| <b>Optional in...</b>  |                            | IPD         |
|                        | Smart Appliance            |             |

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 Section 41.12), 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 Section 41.2). 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 cancellation (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 Section 41.5.1 and Section 41.5.2 respectively. Canceling LCEs is described in Section 41.5.3.

**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.sCancelLoadControlEvents` 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.LoadControlEvents` 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.LoadControlEvents` 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.sCancelLoadControlEvents` 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 `tsSE_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
- *pu8AttributeControlBitsPointer* to an array of **uint8** values, with one element for each attribute in the cluster (see above).
- *psClusterInstancePointer* 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.
- *psClusterDefinitionPointer* 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.
- *psCustomDataStructurePointer* 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
- *pvEndPointSharedStructPtrPointer* 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 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
- *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
- *ppsLoadControlEventPointer* 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
- *ppsLoadControlEventPointer* 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_EXISTS      | 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_ACTIVE  | 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_FOUND       | 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_DEFERRED       | Cancellation has been processed but is deferred to act in the future            |
| E_SE_DRLC_BAD_DEVICE_CLASS      | Specified device class not recognized                                           |
| E_SE_DRLC_BAD_CRITICALITY_LEVEL | Specified criticality level not recognized                                      |
| E_SE_DRLC_DURATION_TOO_LONG     | 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_RES1                | Reserved for future use                                                                                                                                                                                                                                  |
| E_SE_DRLC_GRE1                | <b>Green:</b> Indicates that there will be a significant contribution from non-green sources during the LCE - participation in the LCE is voluntary                                                                                                      |
| E_SE_DRLC_VOL1                | <b>Voluntary 1-6:</b> 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_VOL2                |                                                                                                                                                                                                                                                          |
| E_SE_DRLC_VOL3                |                                                                                                                                                                                                                                                          |
| E_SE_DRLC_VOL4                |                                                                                                                                                                                                                                                          |
| E_SE_DRLC_VOL5                |                                                                                                                                                                                                                                                          |
| E_SE_DRLC_VOL6                |                                                                                                                                                                                                                                                          |
| E_SE_DRLC_EME1                | <b>Emergency:</b> 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_PLA1                | <b>Planned Outage:</b> 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_SERV1               | <b>Service Disconnect:</b> 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_UTII1               | <b>Utility-defined 1-6:</b> Criticality levels completely defined by the utility company - participation in the LCE is voluntary                                                                                                                         |
| E_SE_DRLC_UTII2               |                                                                                                                                                                                                                                                          |
| E_SE_DRLC_UTII3               |                                                                                                                                                                                                                                                          |
| E_SE_DRLC_UTII4               |                                                                                                                                                                                                                                                          |
| E_SE_DRLC_UTII5               |                                                                                                                                                                                                                                                          |
| E_SE_DRLC_UTII6               |                                                                                                                                                                                                                                                          |
| 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_DRLLCCancelControl` 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_CANCEL_IMMEDIATELY | 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_CANCEL_DELAYED     | 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 Section 41.2) |

**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_EVENT_USER_OPT_OUT  | 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_EVENT_USER_OPT_IN   | 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_LIST_SCHEDULED   | <b>Scheduled list:</b> Contains LCEs that are due to be executed in the future                                                              |
| E_SE_DRLC_EVENT_LIST_ACTIVE      | <b>Active list:</b> Contains LCEs that are currently being executed                                                                         |
| E_SE_DRLC_EVENT_LIST_CANCELLED   | <b>Cancelled list:</b> 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_LIST_DEALLOCATED | <b>Deallocated list:</b> 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
} tsSE_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_CANCELLED                     | LCE has been cancelled                                                                           |
| E_SE_DRLC_EVENT_HAS_BEEN_SUPERSEDED                    | LCE has been replaced with another LCE                                                           |
| E_SE_DRLC_EVENT_PARTIALLY_COMPLETED                    | LCE has prematurely completed due to a client opt-out during the LCE                             |
| E_SE_DRLC_EVENT_PARTIALLY_COMPLETED                    | 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_COMMAND              | Received 'cancel command' invalid and rejected (default)                                         |
| E_SE_DRLC_REJECTED_INVALID_CANCEL_COMMAND              | Received 'cancel command' rejected due to invalid effective time (start-time of cancellation)    |
| E_SE_DRLC_REJECTED_EVENT_RECEIVED_AFTER_IT_HAD_EXPIRED | LCE was received after it had expired (current time is greater than start-time + duration)       |
| E_SE_DRLC_REJECTED_INVALID_CANCEL_COMMAND              | 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 0.01°C, for a heating device, where a negative temperature is represented in two's complement form (for example, a temperature of 25°C is represented by 0x09C4 and -1°C 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).
- `u32StartTime` represents the start-time (UTC) of the LCE (although the actual start-time is later if a randomized start-time is required). The value 0x00000000 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 cancellation 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 cancellation 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 cancellation 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
- `u32EventStatusTime` is the time (UTC) at which the Report Event Status message was issued
- `sSignature` 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 |
|------------------------|-----------------|-------------|
| <b>Mandatory in...</b> | Metering Device |             |
| <b>Optional in...</b>  | 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) */
 uint48 u48CurrentSummationDelivered; /
 * Mandatory */
 #ifdef CLD_SM_ATTR_CURRENT_SUMMATION_RECEIVED
 uint48 u48CurrentSummationReceived;
 #endif
 #ifdef CLD_SM_ATTR_CURRENT_MAX_DEMAND_DELIVERED
 uint48 u48CurrentMaxDemandDelivered;
 #endif
 #ifdef CLD_SM_ATTR_CURRENT_MAX_DEMAND_RECEIVED
 uint48 u48CurrentMaxDemandReceived;
 #endif
 #ifdef CLD_SM_ATTR_DFT_SUMMATION
 uint48 u48DFTSummation;
 #endif
 #ifdef CLD_SM_ATTR_DAILY_FREEZE_TIME
 uint16 u16DailyFreezeTime;
 #endif
 #ifdef CLD_SM_ATTR_POWER_FACTOR
 int8 i8PowerFactor;
 #endif
 #ifdef CLD_SM_ATTR_READING_SNAPSHOT_TIME
 tuctime utctReadingSnapshotTime;
 #endif
 #ifdef CLD_SM_ATTR_CURRENT_MAX_DEMAND_DELIVERED_TIME
 tuctime utctCurrentMaxDemandDeliveredTime;
 #endif
 #ifdef CLD_SM_ATTR_CURRENT_MAX_DEMAND_RECEIVED_TIME
 tuctime 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
 uint48 u48CurrentBlockPeriodConsumptionDelivered;
 #endif
 #ifdef CLD_SM_ATTR_DAILY_CONSUMPTION_TARGET
 uint24 u24DailyConsumptionTarget;
 #endif
}
```

(continues on next page)

(continued from previous page)

```

#ifndef CLD_SM_ATTR_CURRENT_BLOCK
 zenum8 e8CurrentBlock;
#endif
#ifndef CLD_SM_SUPPORT_GET_PROFILE
#ifndef CLD_SM_ATTR_PROFILE_INTERVAL_PERIOD
 zenum8 eProfileIntervalPeriod;
#endif
#endif
#ifndef CLD_SM_ATTR_INTERVAL_READ_REPORTING_PERIOD
 uint16 u16IntervalReadReportingPeriod;
#endif
#endif // CLD_SM_SUPPORT_GET_PROFILE
#ifndef CLD_SM_ATTR_PREVIOUS_BLOCK_PERIOD_CONSUMPTION_DELIVERED
 zuint48 u48PreviousBlockPeriodConsumptionDelivered;
#endif
#ifndef CLD_SM_ATTR_PRESET_READING_TIME
 uint16 u16PresetReadingTime;
#endif
#ifndef CLD_SM_ATTR_VOLUME_PER_REPORT
 uint16 u16VolumePerReport;
#endif
#ifndef CLD_SM_ATTR_FLOW_RESTRICTION
 uint8 u8FlowRestriction;
#endif
#ifndef CLD_SM_ATTR_SUPPLY_STATUS
 zbmap8 u8SupplyStatus;
#endif
#ifndef CLD_SM_ATTR_CURRENT_INLET_ENERGY_CARRIER_SUMMATION
 zuint48 u48CurrentInletEnergyCarrierSummation;
#endif
#ifndef CLD_SM_ATTR_CURRENT_OUTLET_ENERGY_CARRIER_SUMMATION
 zuint48 u48CurrentOutletEnergyCarrierSummation;
#endif
#ifndef CLD_SM_ATTR_INLET_TEMPERATURE
 int16 i16InletTemperature;
#endif
#ifndef CLD_SM_ATTR_OUTLET_TEMPERATURE
 int16 i16OutletTemperature;
#endif
#ifndef CLD_SM_ATTR_CONTROL_TEMPERATURE
 int16 i16ControlTemperature;
#endif
#ifndef CLD_SM_ATTR_CURRENT_INLET_ENERGY_CARRIER_DEMAND
 zint24 i24CurrentInletEnergyCarrierDemand;
#endif
#ifndef 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) */
#ifndef CLD_SM_ATTR_CURRENT_TIER_1_SUMMATION_DELIVERED
 zuint48 u48CurrentTier1SummationDelivered;
#endif
#ifndef CLD_SM_ATTR_CURRENT_TIER_1_SUMMATION_RECEIVED
 zuint48 u48CurrentTier1SummationReceived;
#endif
#ifndef CLD_SM_ATTR_CURRENT_TIER_2_SUMMATION_DELIVERED
 zuint48 u48CurrentTier2SummationDelivered;
#endif
#ifndef CLD_SM_ATTR_CURRENT_TIER_2_SUMMATION_RECEIVED
 zuint48 u48CurrentTier2SummationReceived;
#endif
#ifndef CLD_SM_ATTR_CURRENT_TIER_3_SUMMATION_DELIVERED

```

(continues on next page)



(continued from previous page)

```

 uint48 u48CurrentTier3SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_3_SUMMATION_RECEIVED
 uint48 u48CurrentTier3SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_4_SUMMATION_DELIVERED
 uint48 u48CurrentTier4SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_4_SUMMATION_RECEIVED
 uint48 u48CurrentTier4SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_5_SUMMATION_DELIVERED
 uint48 u48CurrentTier5SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_5_SUMMATION_RECEIVED
 uint48 u48CurrentTier5SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_6_SUMMATION_DELIVERED
 uint48 u48CurrentTier6SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_6_SUMMATION_RECEIVED
 uint48 u48CurrentTier6SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_7_SUMMATION_DELIVERED
 uint48 u48CurrentTier7SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_7_SUMMATION_RECEIVED
 uint48 u48CurrentTier7SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_8_SUMMATION_DELIVERED
 uint48 u48CurrentTier8SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_8_SUMMATION_RECEIVED
 uint48 u48CurrentTier8SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_9_SUMMATION_DELIVERED
 uint48 u48CurrentTier9SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_9_SUMMATION_RECEIVED
 uint48 u48CurrentTier9SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_10_SUMMATION_DELIVERED
 uint48 u48CurrentTier10SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_10_SUMMATION_RECEIVED
 uint48 u48CurrentTier10SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_11_SUMMATION_DELIVERED
 uint48 u48CurrentTier11SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_11_SUMMATION_RECEIVED
 uint48 u48CurrentTier11SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_12_SUMMATION_DELIVERED
 uint48 u48CurrentTier12SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_12_SUMMATION_RECEIVED
 uint48 u48CurrentTier12SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_13_SUMMATION_DELIVERED
 uint48 u48CurrentTier13SummationDelivered;

```

(continues on next page)



(continued from previous page)

```

#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_13_SUMMATION_RECEIVED
 uint48 u48CurrentTier13SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_14_SUMMATION_DELIVERED
 uint48 u48CurrentTier14SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_14_SUMMATION_RECEIVED
 uint48 u48CurrentTier14SummationReceived;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_15_SUMMATION_DELIVERED
 uint48 u48CurrentTier15SummationDelivered;
#endif
#ifdef CLD_SM_ATTR_CURRENT_TIER_15_SUMMATION_RECEIVED
 uint48 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
 uint24 u24HoursInOperation;
#endif
#ifdef CLD_SM_ATTR_HOURS_IN_FAULT
 uint24 u24HoursInFault;
#endif
/* Formatting attribute set attribute ID's (D.3.2.2.4) */
zenum8 eUnitOfMeasure; /* Mandatory */
#ifdef CLD_SM_ATTR_MULTIPLIER
 uint24 u24Multiplier;
#endif
#ifdef CLD_SM_ATTR_DIVISOR
 uint24 u24Divisor;
#endif
zbmap8 u8SummationFormatting; /* Mandatory */
#ifdef CLD_SM_ATTR_DEMAND_FORMATTING
 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))
 uint48_t 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))
 uint48_t 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))
 uint48_t 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))
 uint48_t 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))
 uint48_t 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))
 uint48_t 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))
 uint48_t 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))
 uint48_t 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))
 uint48_t 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))
 uint48_t 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))
 uint48_t 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))
 uint48_t 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_t u16GenericAlarmMask;
#endif
#ifdef CLD_SM_ATTR_ELECTRICITY_ALARM_MASK
 zbmap32_t u32ElectricityAlarmMask;
#endif
#ifdef CLD_SM_ATTR_PRESSURE_ALARM_MASK
 zbmap16_t u16PressureAlarmMask;
#endif
#ifdef CLD_SM_ATTR_WATER_SPECIFIC_ALARM_MASK
 zbmap16_t u16WaterSpecificAlarmMask;
#endif
#ifdef CLD_SM_ATTR_HEAT_AND_COOLING_ALARM_MASK
 zbmap16_t u16HeatAndCoolingSpecificAlarmMask;
#endif
#ifdef CLD_SM_ATTR_GAS_ALARM_MASK
 zbmap16_t 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 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_CURR
ENT_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
{
 uint48 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.

- 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.

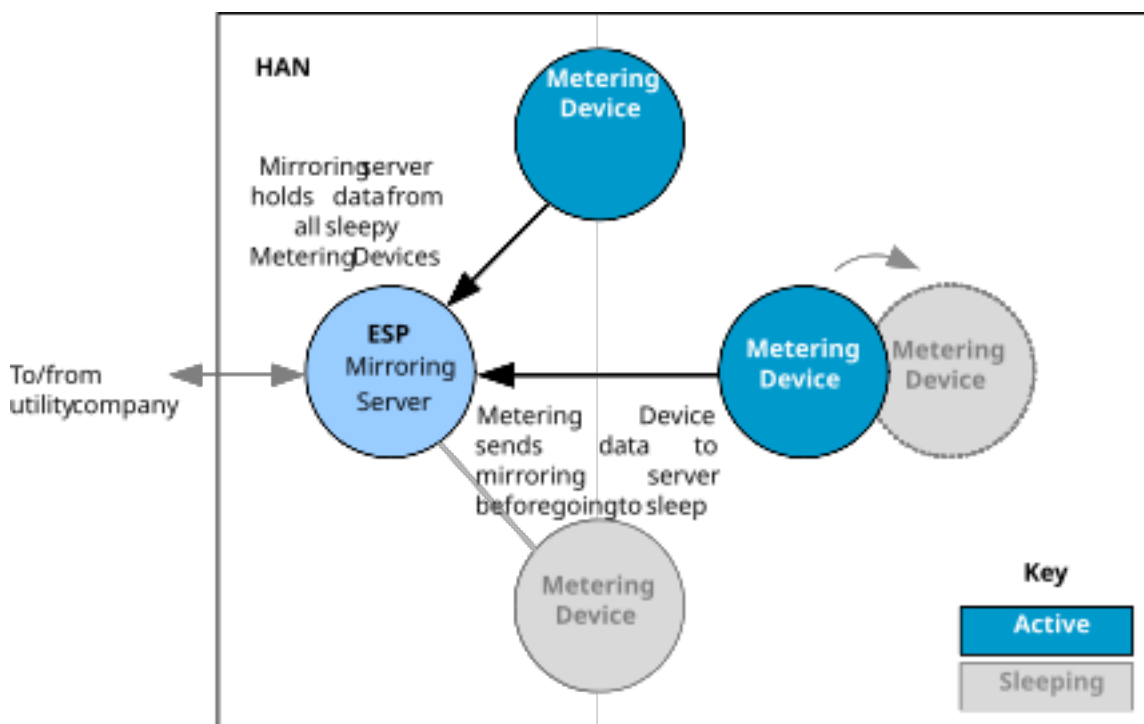
Note that read access to cluster attributes must be explicitly enabled at compile-time as described in Section 1.3.

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.

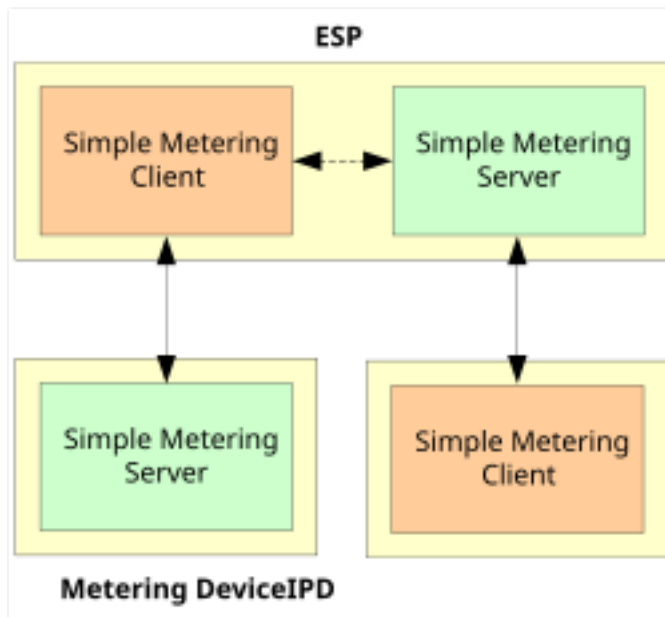
#### Mirroring of Metering Data



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 Section 42.12 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                                                                                    |
|-----------------------------------------|------------------------------------------------------------------------------------------------|
| <code>E_CLD_SM_CLIENT_RECEIVED_C</code> | Generated when a command has been received on a cluster client                                 |
| <code>E_CLD_SM_SERVER_RECEIVED_C</code> | Generated when a command has been received on the cluster server                               |
| <code>E_CLD_SM_FAST_POLLING_TIME</code> | Generated on the cluster server at the end of a fast polling episode ( <i>for future use</i> ) |

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                  |
|-----------------------------------------------|-----------------------------------------|
| <code>E_CLD_SM_GET_PROFILE_RESPONSE</code>    | <code>sGetProfileResponseCommand</code> |
| <code>E_CLD_SM_REQUEST_MIRROR</code>          | <code>sRequestMirrorAdd</code>          |
| <code>E_CLD_SM_REMOVE_MIRROR</code>           | <code>sRequestMirrorRemove</code>       |
| <code>E_CLD_SM_REQUEST_FAST_POLL_MODE_</code> |                                         |

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                                |
|-----------------------------------------------|-------------------------------------------------------|
| <code>E_CLD_SM_GET_PROFILE</code>             | <code>sGetProfileCommand</code>                       |
| <code>E_CLD_SM_REQUEST_MIRROR_RESPONSE</code> | <code>sRequestMirrorResponseCommand</code>            |
| <code>E_CLD_SM_MIRROR_REMOVED</code>          | <code>sMirrorRemovedResponseCommand</code>            |
| <code>E_CLD_SM_REQUEST_FAST_POLL_MODE</code>  | <code>sRequestFastPollCommand</code> (for future use) |
| <code>E_CLD_SM_SERVER_ERROR</code>            | <code>sError</code>                                   |

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\_SMCreat
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\_SMCreat

```
teZCL_Status eSE_SMCreat(
 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 to 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_FORMATTING,
E_CLD_SM_ATTR_ID_DEMAND_FORMATTING,
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                               |
|----------------------------------|-------------------------------------------|
| <b>Instantaneous</b>             |                                           |
| <b>Binary Values</b>             |                                           |
| 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                 |
| <b>BCD Values</b>                |                                           |
| 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_RIGHT        | Position of least significant bit of bit-field indicating number of digits to right of decimal point |
| E_CLD_SM_FORMATTING_DIGITS_TO_RIGHT        | Number of bits in bit-field indicating number of digits to right of decimal point                    |
| E_CLD_SM_FORMATTING_DIGITS_TO_RIGHT        | Bit-mask used to extract number of digits to right of decimal point                                  |
| E_CLD_SM_FORMATTING_DIGITS_TO_LEFT         | Position of least significant bit of bit-field indicating number of digits to left of decimal point  |
| E_CLD_SM_FORMATTING_DIGITS_TO_LEFT         | Number of bits in bit-field indicating number of digits to left of decimal point                     |
| E_CLD_SM_FORMATTING_DIGITS_TO_LEFT         | Bit-mask used to extract number of digits to left of decimal point                                   |
| E_CLD_SM_FORMATTING_SUPPRESS_LEADING_ZEROS | Bit-mask used to extract bit indicating whether leading 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 ( <i>for future use</i> ) |

**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             | Enumera-<br>tion | Description                                                                                                                                                           |
|---------------------|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| E_CLD_SM_GET_PROFI  |                  | Response to 'Get Profile' request - content of response is contained in the structure <code>tsS-M_GetProfileResponseCommand</code> 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 to 'Fast Polling' request ( <i>for future use</i> )                                                                                                          |
| E_CLD_SM_CLIENT_ERR |                  | Error condition - content of error is contained in the structure <code>tsSM_Error</code> 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          | Enu-<br>meration | Description                                                                                                                                                                |
|------------------|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| E_CLD_SM_GET_PRC |                  | A 'Get Profile' request - content of request is contained in the structure <code>tsSM_GetProfile-RequestCommand</code> in the event (see Section 42.11.8)                  |
| E_CLD_SM_REQUES  |                  | Response to 'Add Mirror' request - content of response is contained in the structure <code>tsS-M_RequestMirrorResponseCommand</code> 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 <code>tsS-M_MirrorRemovedResponseCommand</code> in the event (see Section 42.11.7) |
| E_CLD_SM_REQUES  |                  | A 'Fast Polling' request ( <i>for future use</i> )                                                                                                                         |
| E_CLD_SM_SERVER  |                  | Error condition - content of error is contained in the structure <code>tsSM_Error</code> 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_INTERVAL_CHANNEL                | Undefined eIntervalChannel value specified in ‘Get Profile’ request (see Section 42.11.8)                      |
| E_CLD_SM_STATUS_INTERVAL_NOT_SUPPORTED                    | Unsupported consumption data specified through eIntervalChannel in ‘Get Profile’ request (see Section 42.11.8) |
| E_CLD_SM_STATUS_INVALID_END_TIME                          | Invalid end-time specified in ‘Get Profile’ request (Section 42.11.8)                                          |
| E_CLD_SM_STATUS_MORE_PERIODS_REQUESTED_THAN_SUPPORTED     | More periods specified in ‘Get Profile’ request than can be returned                                           |
| E_CLD_SM_STATUS_NO_INTERVALS_AVAILABLE_FOR_REQUESTED_TIME | No intervals available for the end-time specified in ‘Get Profile’ request                                     |
| E_CLD_SM_STATUS_EP_NOT_AVAILABLE                          | 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
{
 tsSM_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 Section 42.10.7
- `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`):
  - `sGetProfileResponseCommand` 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 Section 42.11.1) 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;
 uint24 u24ConsumptionReceived;
 uint24 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 structure 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 structure 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;
 uint24 *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 structure 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\_tsclد\_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\_tsclد\_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\_tsclد\_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\_tsclد\_concentratorparamete.md#id\_3713b9e8-5da6-4db8-b678-617069de5091)

`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).

**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_t u16ShortAddress;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_EXTENED_PAN_ID
 zieeeaddress_t u64ExtPanId;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_PAN_ID
 uint16_t u16PANId;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_CHANNEL_MASK
 zbmap32_t u32ChannelMask;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_PROTOCOL_VERSION
 uint8_t u8ProtocolVersion;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_STACK_PROFILE
 uint8_t u8StackProfile;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_START_UP_CONTROL
 zenum8_t e8StartUpControl;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_TC_ADDR
 zieeeaddress_t u64TcAddr;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_TC_MASTER_KEY
 tsZCL_Key_t sTcMasterKey;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_NWK_KEY
 tsZCL_Key_t sNwkKey;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_USE_INSECURE_JOIN
 bool_t bUseInsecureJoin;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_PRE_CONFIG_LINK_KEY
 tsZCL_Key_t sPreConfigLinkKey;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_NWK_KEY_SEQ_NO
 uint8_t u8NwkKeySeqNo;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_NWK_KEY_TYPE
 zenum8_t e8NwkKeyType;
 #endif
 #ifdef CLD_COMMISSIONING_ATTR_NWK_MANAGER_ADDR
 uint16_t 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-

indicates 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_t u16IndirectPollRate;
#endif
#ifdef CLD_COMMISSIONING_ATTR_PARENT_RETRY_THRSHLD
 uint8_t 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_t u8ConcentratorRadius;
#endif
#ifdef CLD_COMMISSIONING_ATTR_CONCENTRATOR_DISCVRY_TIME
 uint8_t 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** 0. 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 Section 43.9.6).

```
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_DEVICE  | 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_STARTUP | A Restore Start-up Parameters command has been received |
| E_CLD_COMMISSIONING_CMD_RESET_STARTUP   | 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_DEVICE  | 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_STARTUP | A Restore Start-up Parameters response has been received |
| E_CLD_COMMISSIONING_CMD_RESET_STARTUP   | 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

```
tsZCL_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:

| Com-mand                     | 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 <b>eCLD_CommissioningCommandRestartDeviceSend()</b> . 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 <b>eCLD_CommissioningCommandSaveStartupParamsSend()</b> .                                                                                                                                                                    |
| Re-store 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 values, as described for the function <b>eCLD_CommissioningCommandRe-storeStartupParamsSend()</b> . 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 <b>eCLD_CommissioningCommandResetStartupParamsSend()</b> . 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_PA  | 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 following 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
{
 zbm8 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 specified 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 Section 43.9.3
- `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 Utility 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**



| Com-mand                             | Iden-ti-fier | Description                                                                                                                                                                                                                                                                                                                                                              |
|--------------------------------------|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Scan Re-request *                    | 0x0C         | 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 Re-request *      | 0x02         | Requests information about the devices on a remote node                                                                                                                                                                                                                                                                                                                  |
| Identify Re-request                  | 0x06         | Requests a remote node to physically identify itself (for example, visually by flashing an LED)                                                                                                                                                                                                                                                                          |
| Reset To Factory New Re-request      | 0x07         | Requests a factory reset of a remote node                                                                                                                                                                                                                                                                                                                                |
| Network Start Re-request *           | 0x1C         | Requests a new network to be created comprising the initiator and a detected Router                                                                                                                                                                                                                                                                                      |
| Network Join Router Re-request *     | 0x12         | Requests a Router to join the network                                                                                                                                                                                                                                                                                                                                    |
| Network Join End Device Re-request * | 0x14         | Requests an End Device to join the network                                                                                                                                                                                                                                                                                                                               |
| Network Update Re-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

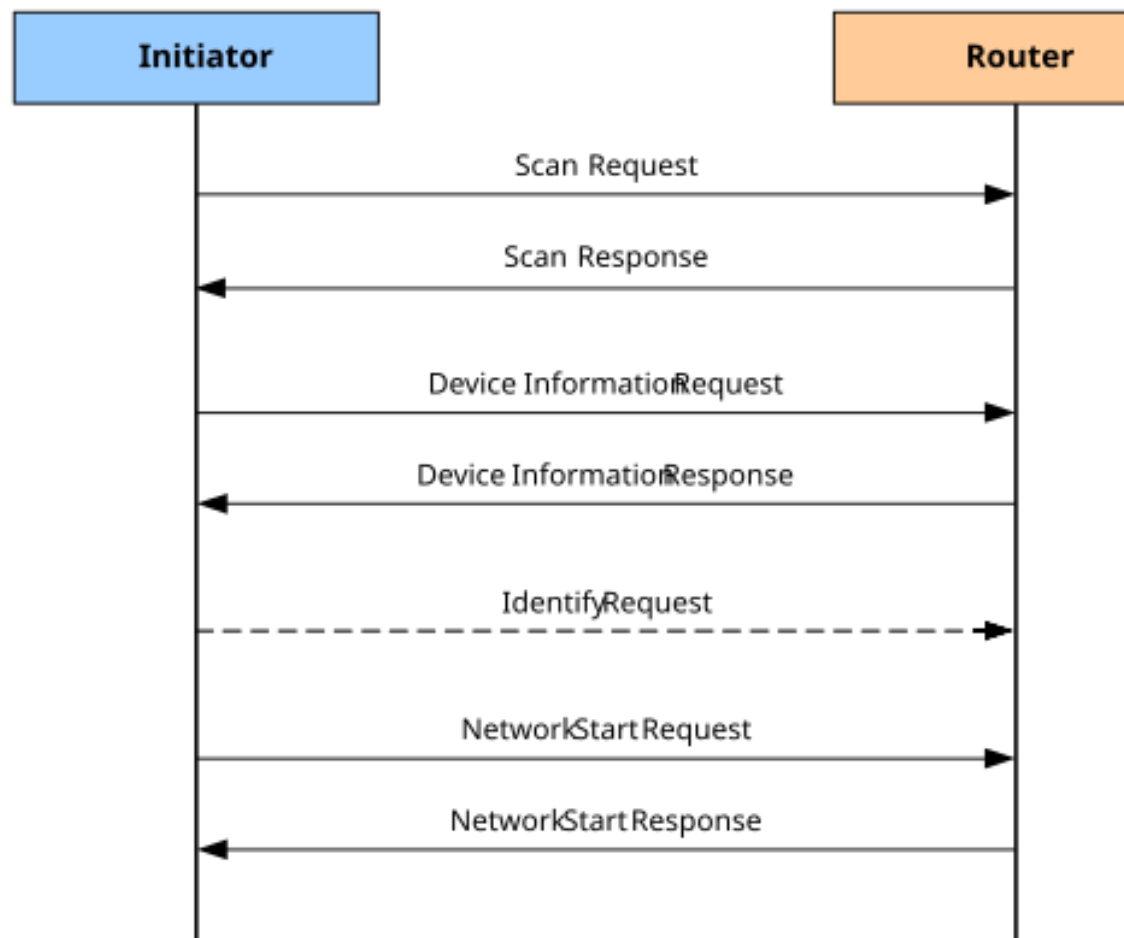
the command list in the table in the Section Using Touchlink).

**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 initiator 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.

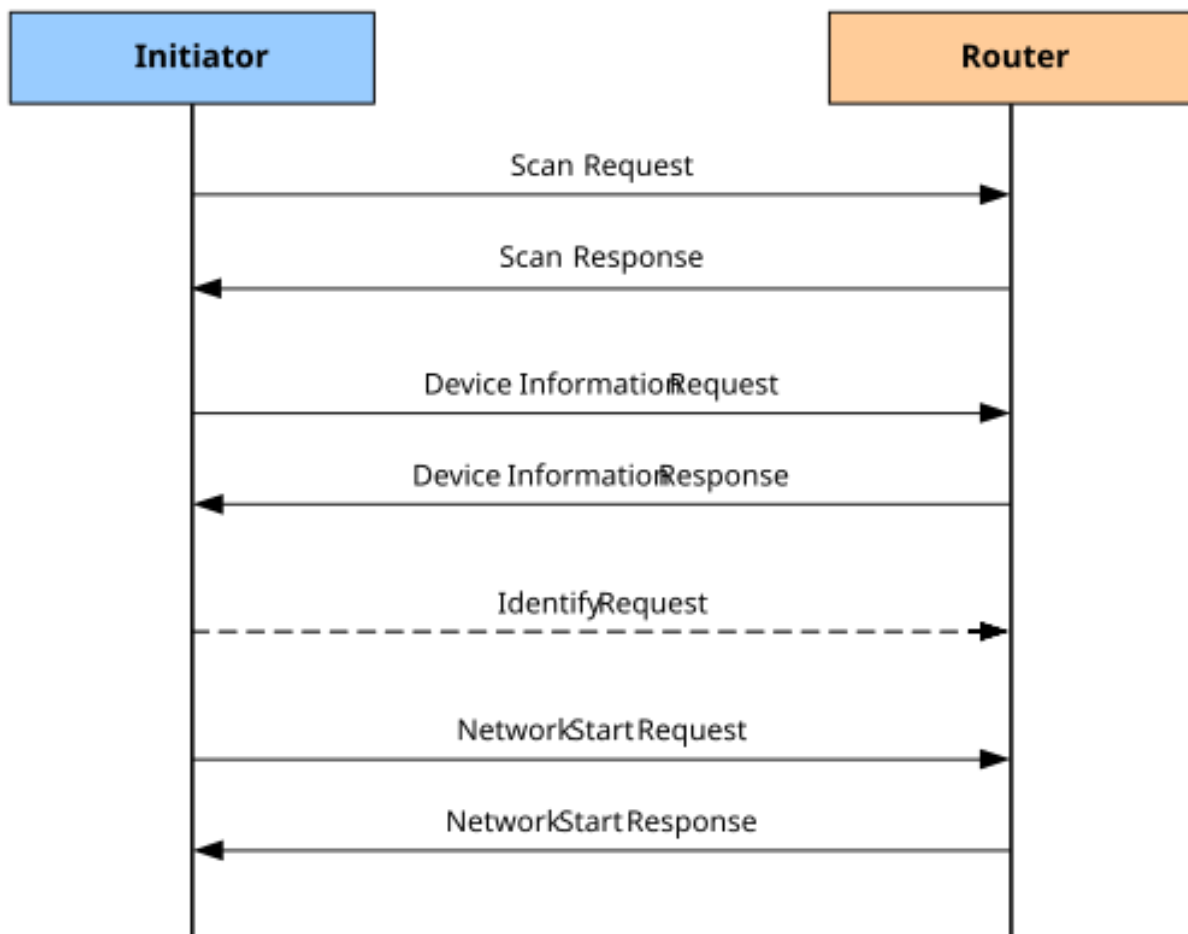
1. **Scan Request:** The initiator sends a Scan Request to nodes in its vicinity. The required function is:
  - **eCLD\_ZllCommissionCommandScanReqCommandSend()**
2. **Scan Response:** A receiving node replies to the Scan Request by sending a Scan Response. The required function is:
  - **eCLD\_ZllCommissionCommandScanRspCommandSend()**
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) |

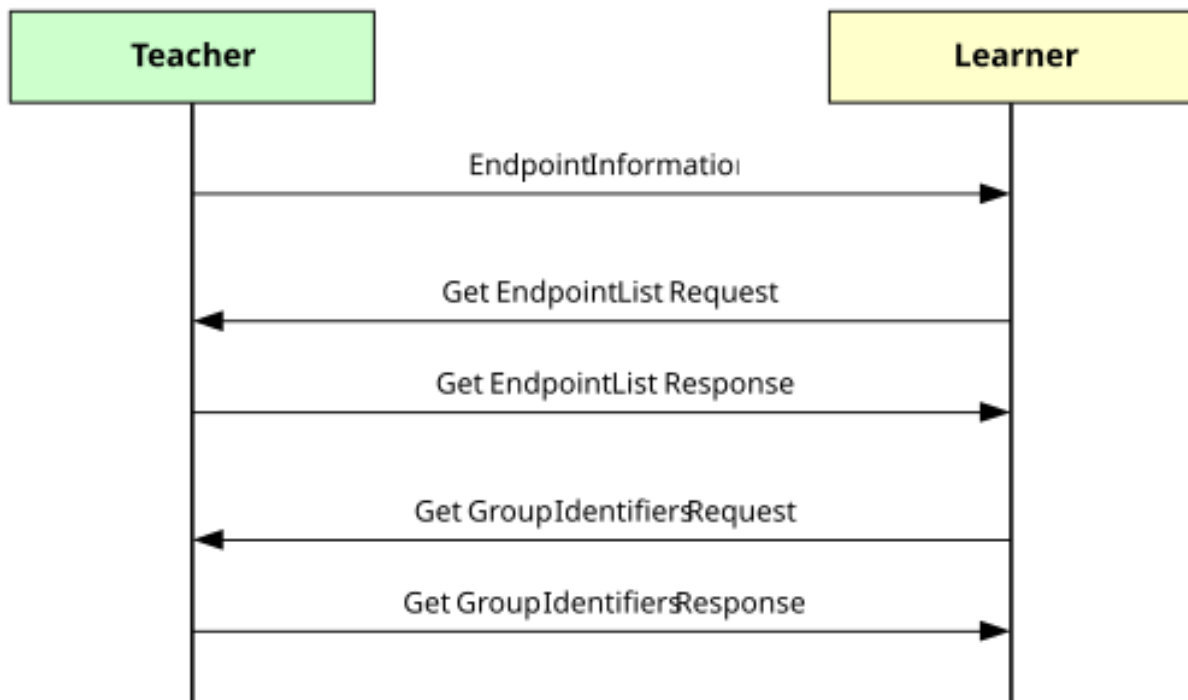
Use of the above commands and associated functions is described below and is illustrated in the figure below.

**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                                                  |
|------------------------------------------------|--------------------------------------------------------------|
| <code>E_CLD_COMMISSION_CMD_SCAN_REQ</code>     | A Scan Request has been received (by server)                 |
| <code>E_CLD_COMMISSION_CMD_SCAN_RSP</code>     | A Scan Response has been received (by client)                |
| <code>E_CLD_COMMISSION_CMD_DEVICE_INFO</code>  | A Device Information Request has been received (by server)   |
| <code>E_CLD_COMMISSION_CMD_DEVICE_INFO</code>  | A Device Information Response has been received (by client)  |
| <code>E_CLD_COMMISSION_CMD_IDENTIFY_REQ</code> | An Identify Request has been received (by server)            |
| <code>E_CLD_COMMISSION_CMD_FACTORY_RES</code>  | A Reset To Factory New Request has been received (by server) |
| <code>E_CLD_COMMISSION_CMD_NETWORK_ST</code>   | A Network Start Request has been received (by server)        |
| <code>E_CLD_COMMISSION_CMD_NETWORK_ST</code>   | A Network Start Response has been received (by client)       |
| <code>E_CLD_COMMISSION_CMD_NETWORK_JOIN</code> |                                                              |

#### 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_G | 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 structure 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 Section 44.8.5)

### 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 structure 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 structure 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 structure 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 structure 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 structure 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 structure 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 structure 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 structure 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 structure 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_t 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 structure 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 structure 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 structure 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 structure 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 structure 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 structure 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 structure 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 structure 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;
 #endif
}
```

(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:
  - `sZllCommissionClientCluster` is 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 Confidentiality Agreement')
  - 0x8000 (bit 15 set): Certification key (defined in the ZLL Specification for use during development and during certification at test houses)
- `u32ResponseId` 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_t u32TransactionId;
 uint8_t 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_t u32TransactionId;
 uint8_t u8NumberSubDevices;
 uint8_t u8StartIndex;
 uint8_t 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_t 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
    - 0x0001–0xFFFFE: 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_t u32TransactionId;
 uint8_t u8Status;
 uint64_t u64ExtPanId;
 uint8_t u8NwkUpdateId;
 uint8_t u8LogicalChannel;
 uint16_t 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_t u32TransactionId;
 uint8_t 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_t u32TransactionId;
 uint64_t u64ExtPanId;
 uint8_t u8KeyIndex;
 uint8_t 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_t u32TransactionId;
 uint8_t 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_t u32TransactionId;
 uint64_t u64ExtPanId;
 uint8_t u8NwkUpdateId;
 uint8_t u8LogicalChannel;
 uint16_t u16PanId;
 uint16_t 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
 uint16_t u16StartTime;
 uint16_t u16FinishTime;
#ifdef CLD_APPLIANCE_CONTROL_REMAINING_TIME
 uint16_t u16RemainingTime;
#endif
#ifdef CLD_APPLIANCE_CONTROL_ATTRIBUTE_REPORTING_STATUS
 uint8_t u8AttributeReportingStatus;
#endif
#endif
 uint16_t 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 operation 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 appliance |

| 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 appliance 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 Section 7.2.

**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_RESPON | 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 eEvent-Type 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
{
 uint16_t 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                                                        |
|--------------------------------------------------|--------------------------------------------------------------------|
| <code>E_CLD_APPLIANCE_EVENTS_AND_ALERTS_C</code> | A ‘Get Alerts’ request has been received by the server (appliance) |

| u8CommandId Enumeration                    | Description                                                                                                      |
|--------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| <code>E_CLD_APPLIANCE_EVENTS_AND_AL</code> | A response to a ‘Get Alerts’ request has been received by the client, containing the requested alerts (up to 15) |
| <code>E_CLD_APPLIANCE_EVENTS_AND_AL</code> | An ‘Alerts Notification’ message has been received by the client, containing unsolicited alerts (up to 15)       |
| <code>E_CLD_APPLIANCE_EVENTS_AND_AL</code> | 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\_AEAAAAlertsNotificationSend
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
{
 uint8_t u8AlertsCount;
 uint24_t 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
{
 uint8_t u8EventHeader;
 uint8_t 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
 uint32_t u32LogMaxSize;
 uint8_t u8LogQueueMaxSize;
 #endif
 uint16_t 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 `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
 *psLogQueueResponseORStatisticsAvailablePayload;
 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 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_RESPONSE   | A 'Log Queue Response' message   |
| E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE | 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
 *psLogQueueResponseORStatisticsAvailablePayload;
 tsCLD_ASC_LogRequestPayload *psLogRequestPayload;
 } uMessage;
} tsCLD_ApplianceStatisticsCallbackMessage;
```

where:

- `u8CommandId` 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`
- `uMessage` 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 Section 48.9.3)
  - `psLogQueueResponseORStatisticsAvailablePayload` 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
{
 uint32_t 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
{
 time_t utctTime;
 uint32_t u32LogId;
 uint32_t u32LogLength;
 uint8_t *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
{
 uint8_t u8LogQueueSize;
 uint32_t *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
{
 utctime 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:

|                 |               |   |   |   |   |       |   |   |           |   |    |    |    |       |    |    |
|-----------------|---------------|---|---|---|---|-------|---|---|-----------|---|----|----|----|-------|----|----|
| 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 |    |    |

#### Case 1 (Sector 8)

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:

|                 |                   |   |    |    |    |       |   |   |           |   |    |    |    |       |    |    |
|-----------------|-------------------|---|----|----|----|-------|---|---|-----------|---|----|----|----|-------|----|----|
| 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 |    |    |

#### Case 2

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).

|                 |               |   |    |    |    |       |    |    |           |   |    |    |    |    |       |    |
|-----------------|---------------|---|----|----|----|-------|----|----|-----------|---|----|----|----|----|-------|----|
| 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 |    |

#### Case 3

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:

|                 |               |   |   |   |   |       |   |   |           |   |    |    |    |    |       |    |
|-----------------|---------------|---|---|---|---|-------|---|---|-----------|---|----|----|----|----|-------|----|
| 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 |    |

#### Case 4

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):

|                 |               |   |   |   |   |       |   |   |           |   |    |    |    |    |       |      |
|-----------------|---------------|---|---|---|---|-------|---|---|-----------|---|----|----|----|----|-------|------|
| 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 |

#### Case 5

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:

|                 |                   |   |   |    |    |       |    |           |   |   |    |    |       |    |      |    |
|-----------------|-------------------|---|---|----|----|-------|----|-----------|---|---|----|----|-------|----|------|----|
| 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 |    |

**Case 6**

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 u64UpgradeServerID;
 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:

- `u64UpgradeServerID` 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.

| <code>u8ImageUpgradeStatus</code> | 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.

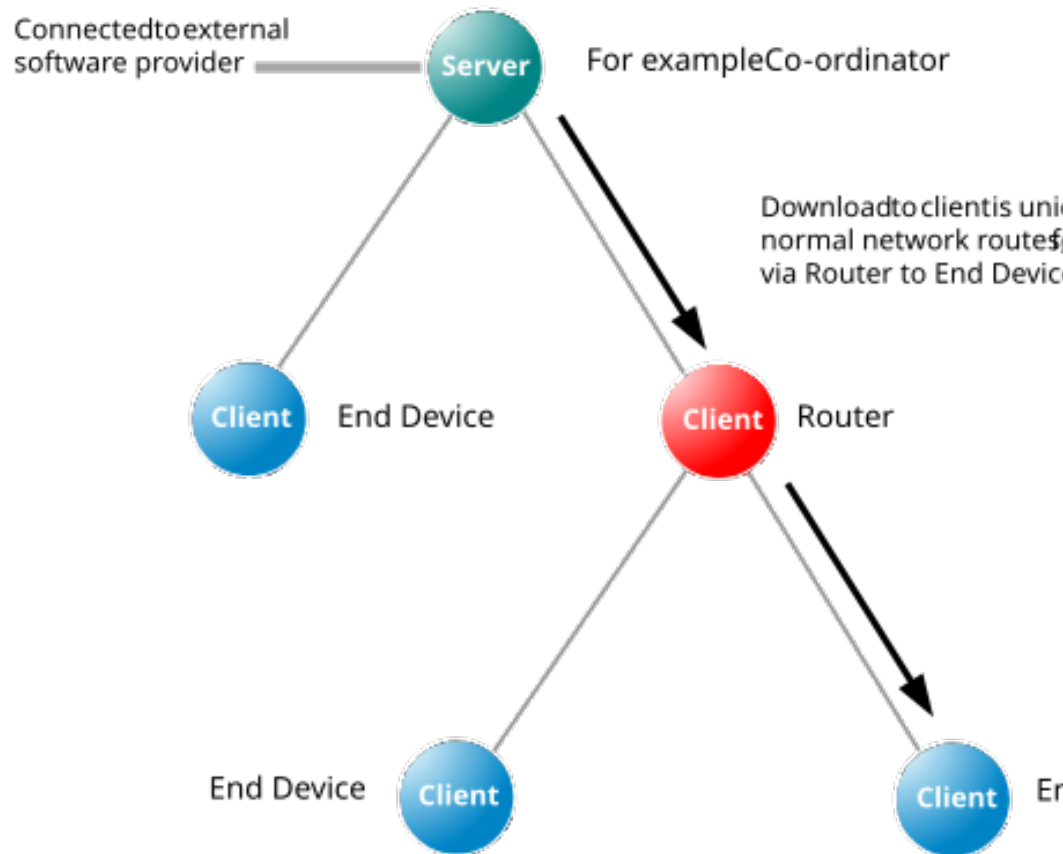
Thus, the OTA Upgrade cluster structure contains only two mandatory elements, `u64UpgradeServerID` and `u8ImageUpgradeStatus`. The remaining elements are optional, each being enabled/disabled through a corresponding macro defined in the `zcl_options.h` file (see Section 49.13).

**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\_eAplAflnit()** 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 is 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:

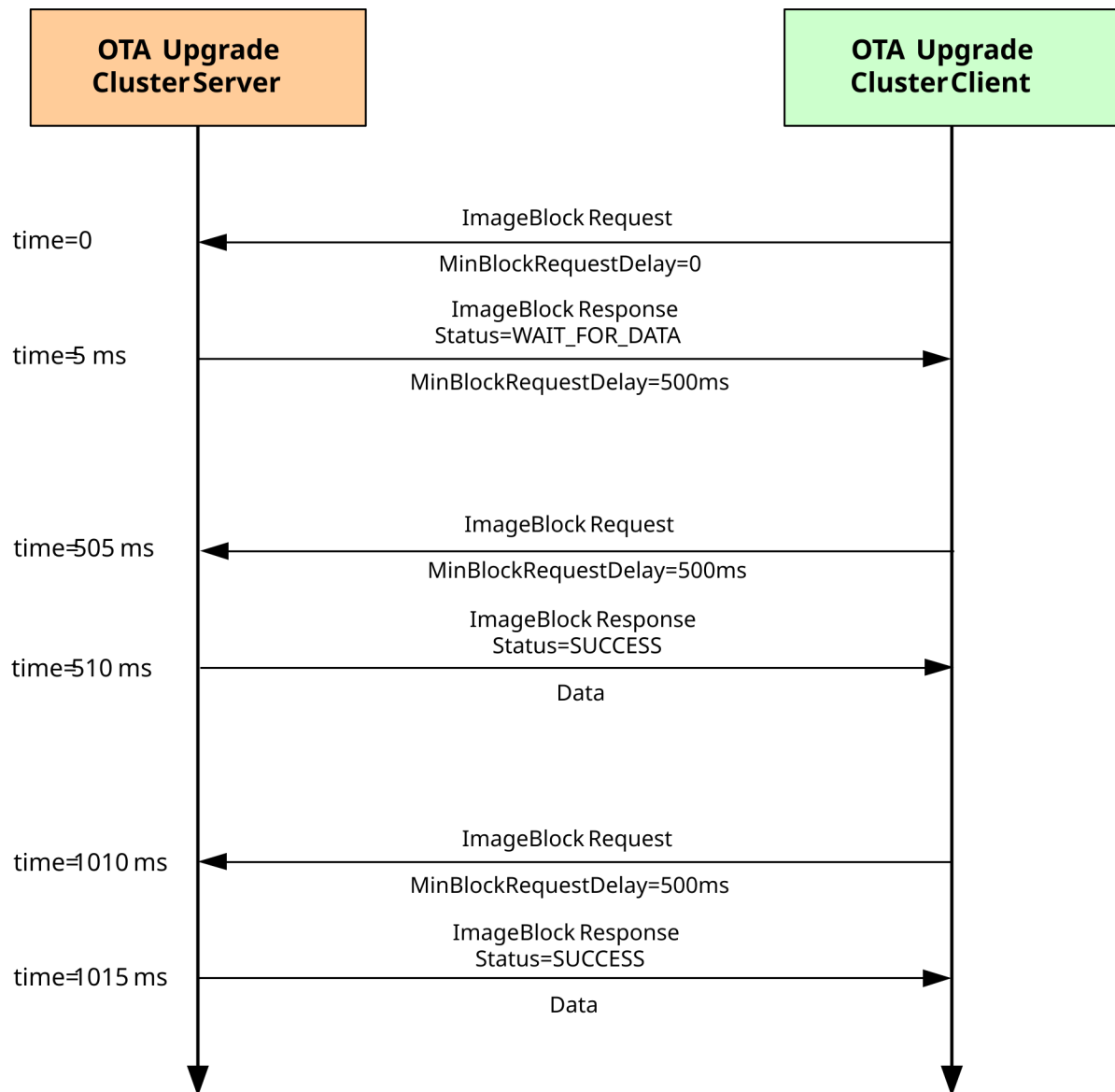
```
sZCL_CallbackEvent.uMessage.u32TimerPeriodMs
```

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 Upgrade 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 <b>eOTA_ClientQuerySpecificFileRequest()</b> .                                                                                   |                                                                                                                                                                                  |
| 2                                                                                                                                                                                                                         | On arrival at the server, the Query Specific File Request triggers the event <b>E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_REQUEST</b> .                                              |
| 3                                                                                                                                                                                                                         | Server automatically replies to the request with a Query Specific File Response - the application can also send a response using <b>eOTA_ServerQuerySpecificFileResponse()</b> . |
| 4 On arrival at the client, the Query Specific File Response triggers the event <b>E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_RESPONSE</b> .                                                                                   |                                                                                                                                                                                  |
| 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 <b>E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_COMPLETE</b> .                                                                                 |                                                                                                                                                                                  |
| 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_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_INTERNAL_COMMAND_QUERY_NEXT_IMAGE_RESPONSE             |
| Client          | 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_BLOCK_RESPONSE_TAG_OTHER_THAN_UPGRADE_IMAGE                   | E_CLD_OTA_IN |
| 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_CallbackEvent` structure 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,
 AEISSW_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 device's 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 Section 49.11.15)



## 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 be 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 Section 49.11.11)

### 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 *psOTADData,
 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

- *u8Endpoint*: Number of endpoint corresponding to context data
- *psOTADData*: 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__HwFnCbTable 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 Section 49.11.15)

**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
#ifdef (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 u16CurrentZigbeeStackVersion;
}tsOTA_QuerySpecificFileRequestPayload;
```

where:

- `u64RequestNodeAddress` is the IEEE/MAC address of the node requesting the device-specific file from the server
- `u16ManufacturerCode` 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)*
- `u16CurrentZigBeeStackVersion` 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)
- `u16ManufacturerCode` 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 au8ReadOTADData[OTA_MAX_BLOCK_SIZE];
 uint8 u8NextFreeImageLocation;
 uint8 u8CurrentActiveImageLocation;
#endif
#ifdef OTA_SERVER
 tsCLD_PR_Ota aServerPrms[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;
#endif
#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 Section 49.12.2)
- 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
- au8ReadOTADData 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
- aServerPrms 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 device's 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)
- u8QueryJitter is 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 u64UpgradeServerID;
 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\_acc84e1-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_VERSION | 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_COMMAND_NOTIFY_IMAGE_AVAILABLE           | 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_COMMAND_QUERY_NEXT_IMAGE_REQUEST         | 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_COMMAND_QUERY_NEXT_IMAGE_RESPONSE        | 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_COMMAND_REQUEST_IMAGE_BLOCK              | 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_COMMAND_REQUEST_IMAGE_PAGE               | 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_COMMAND_RESPONSE_IMAGE_BLOCK             | 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_COMMAND_UPGRADE_END_REQUEST              | 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_COMMAND_UPGRADE_END_RESPONSE             | 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_COMMAND_QUERY_SPECIFIC_FILE_REQUEST      | Generated on server when a Query Specific File Request is received from a client to request a particular application image                                                                                       |
| E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_RESPONSE     | 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_INTERNAL_COMMAND_NOTIFY_EXPIRED_TIMER    | Generated on client to notify the application that the local one-second timer has expired                                                                                                                        |
| E_CLD_OTA_INTERNAL_COMMAND_PROMPT_STORE_CONTEXT    | Generated on server or client to prompt the application to store context data in Flash memory                                                                                                                    |
| E_CLD_OTA_INTERNAL_COMMAND_ABORT_DOWNLOAD          | 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_INTERNAL_COMMAND_PROMPT_POLL_SERVER      | Generated on client to prompt the application to poll the server for a new application image                                                                                                                     |
| E_CLD_OTA_INTERNAL_COMMAND_NOTIFY_RESET_DEVICE     | Generated on client to notify the application that the stack is going to reset the device                                                                                                                        |
| E_CLD_OTA_INTERNAL_COMMAND_PROMPT_LOCK_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_LIST | 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_FVERSION | 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 (NUG3130)*.

**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 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 Zig-Bee 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_DeviceTemperatureConfigurationCreateDeviceTemperatureConfiguration()`
- 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\_SMCreat()**
- 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
 uint16 u16ACFrequency;
#endif
#ifdef CLD_ELECTMEAS_ATTR_RMS_VOLTAGE
 uint16 u16RMSVoltage;
#endif
#ifdef CLD_ELECTMEAS_ATTR_RMS_CURRENT
 uint16 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
 uint16 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)(amp((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)(amp((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,
 GEN-
 ERAL_CLUSTER_ID_BASIC,
 TRUE,
 E_CLD_BASIC_CMD_MANU_SPEC,
 pu8TransactionSe-
 asPayloadDefinition,
 TRUE,
 HA_MANUFACTURER_CODE,
 sizeof(asPayloadDefinition) / sizeof(tsZCL_TxPayloadItem));
 return eZCL_Status;
}

```

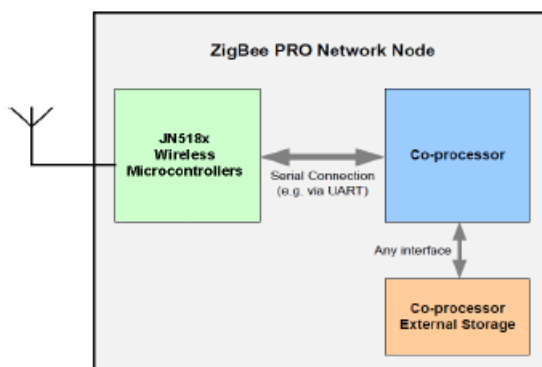
\*\*Parent topic:\*\*[Appendix E: Manufacturer-specific attributes and commands](../appendix/topics/manufacturer-specific\_attributes\_and\_commands.md)

**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.



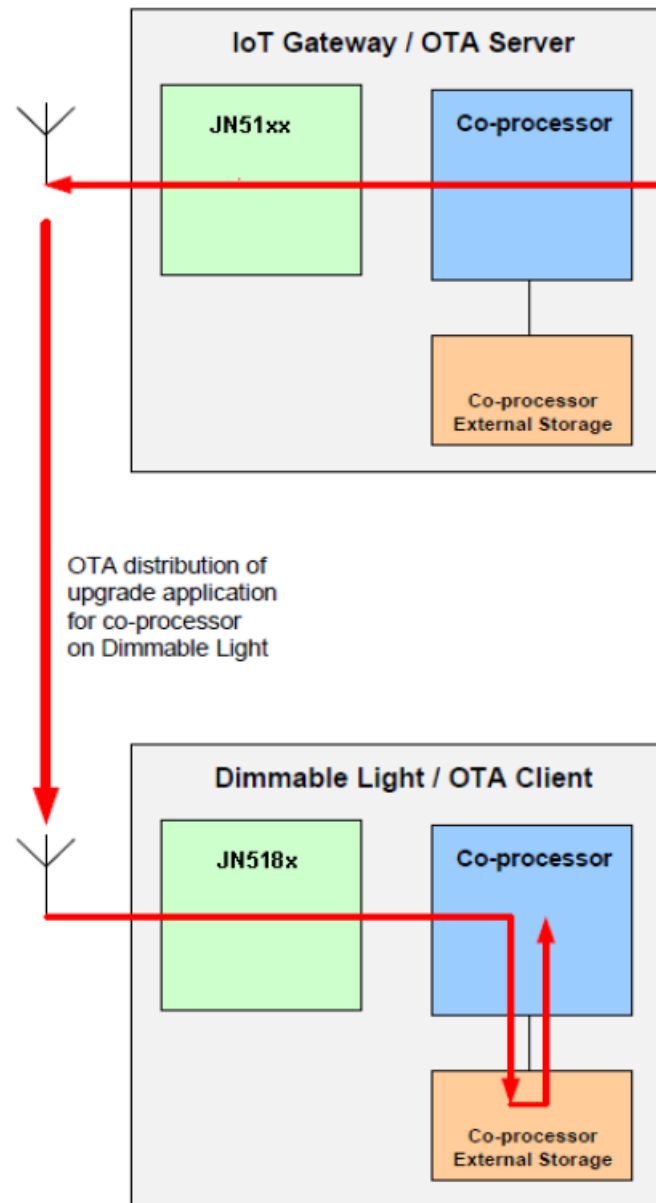
### Dual-Processor Node

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 concerned        |
| 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-wo      |
| 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 transi       |
| 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-leve    |
| 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-tional     |
| 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

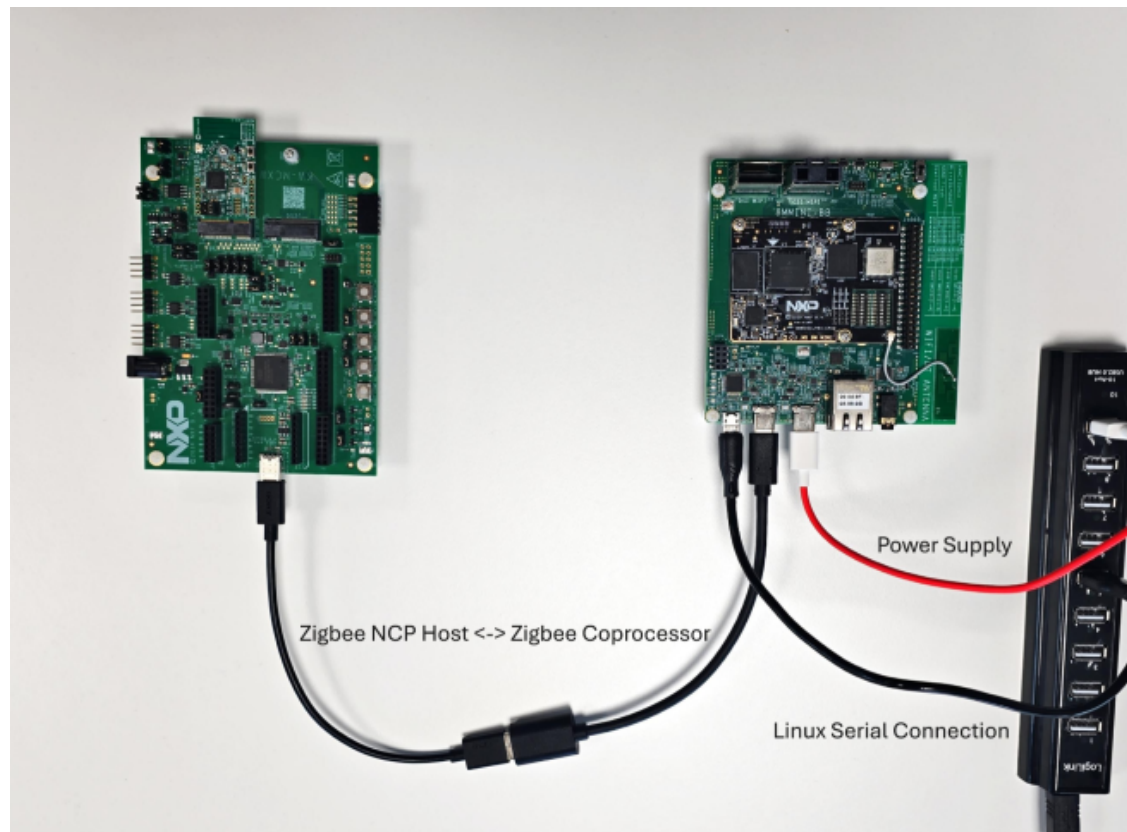
### 2.1. iMX8 board configuration

### 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 `mbedtls` 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 13.2.1
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/aarch64-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-linux-gnu-g++ - 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 CMakeFiles/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 nphost-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 CMakeFiles/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 Profile 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

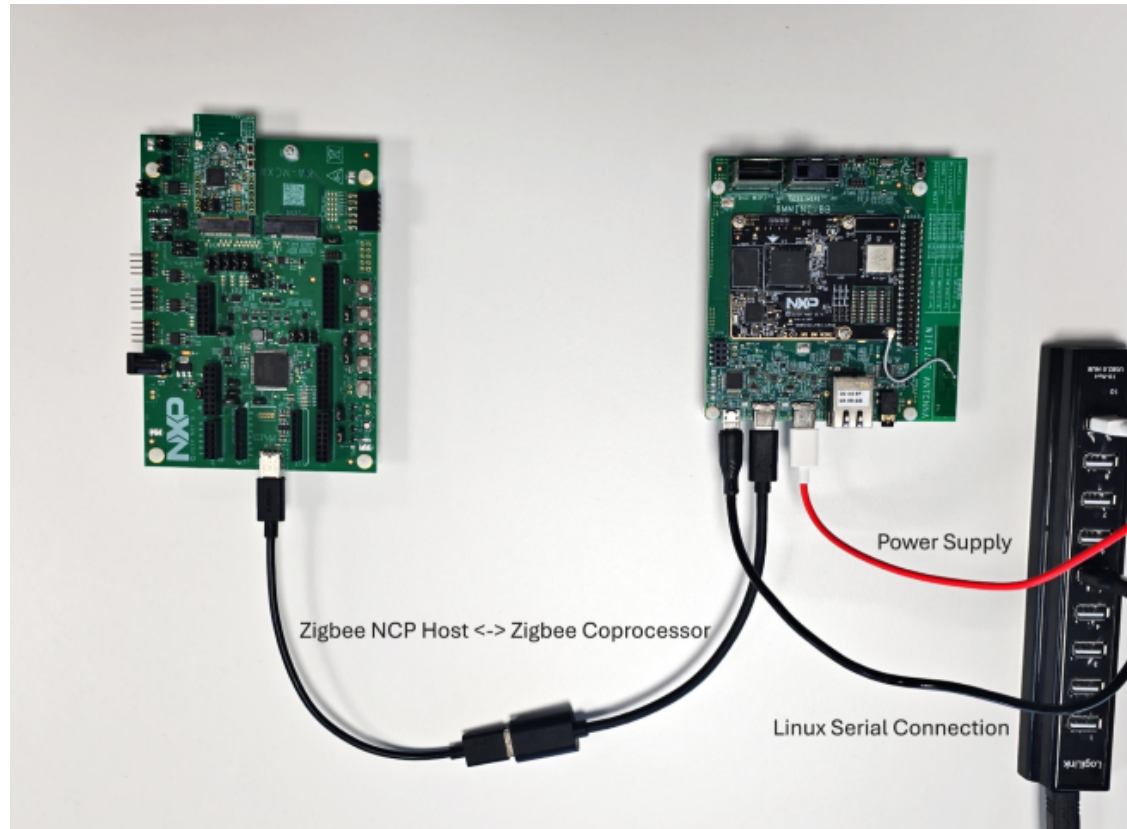
### 2.1. iMX8 board configuration

### 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 `ZIG-BEE_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 `mbedtls` 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 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 CMakeFiles/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 ncp-host-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 CMakeFiles/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 Profile 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:

- MCX-W71-EVK and FRDM-MCXW71 boards require `k32w1_mcxw71_nbu_ble_15_4_dyn_sp.sb3`
- MCX-W72-EVK and FRDM-MCXW72 boards require `mcxw72_nbu_ble_full_15_4_dyn.bin`

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                                                                                                                                                |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>zigbee_factoryreset</code>       | Resets the device to its factory state                                                                                                                     |
| <code>zigbee_get_network_status</code> | Returns the network status <code>NETWORK_CREATED</code> or <code>NETWORK_NOT_CREATED</code>                                                                |
| <code>zigbee_get_network_info</code>   | Returns the network info, Network Info would be serialized in the TLV format if created or <code>NETWORK_NOT_CREATED</code> (if the network is not formed) |
| <code>zigbee_create_network</code>     | Allows to create a zigbee network, <code>NETWORK_ALREADY_CREATED</code> will be returned if the network is already formed                                  |
| <code>zigbee_find</code>               | Start Find & Bind as an initiator, make sure to trigger Find & Bind on a TARGET device                                                                     |
| <code>zigbee_toggle</code>             | 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:

- MCX-W71-EVK and FRDM-MCXW71 boards require `k32w1_mcxw71_nbu_ble_15_4_dyn_sp.sb3`
- MCX-W72-EVK and FRDM-MCXW72 boards require `mcxw72_nbu_ble_full_15_4_dyn.bin`

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                                                                                                    |
|------------------------|----------------------------------------------------------------------------------------------------------------|
| zig-bee_factoryreset   | Resets the device to its factory state                                                                         |
| zig-bee_get_network_s  | Returns the network status CONNECTED or NOT_CONNECTED                                                          |
| zig-bee_get_device_inf | Returns the device info (extended address, installation code, CRC), that would be serialized in the TLV format |
| zig-bee_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](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
- HWPParameters: 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

### 7.0.3 revA mcux SDK 25.06.00

#### 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\_HSL0T) 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\_Get3KTimeStamp() 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.
- [HWPParameter][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 gRng-MaxRequests\_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 algorithm 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_config.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 `gHwParamsProdDataMainFlash2IfirMode_c` to `gHwParamsProdDataIfirMode_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, frdm-mcxw72 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 variable. Allow the application to override the trig sample number parameter when gFsciOverRpmMsg\_c is set to 1
- [DBG][btsnoop] Add a mechanism to dump raw HCI data via UART using SBT-SNOOP\_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.
    - \* Explicated RNG error codes
    - \* Added argument checks for all APIs and return gRngBadArguments\_d (-2) when wrong

- \* added checks of RNG initialization and return `gRngNotInitialized_d (-3)` when not done
- \* fixed correctness 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.
- [SecLib\_RNG] Use defragmentation option when freeing key object in `SecLib_sss` to avoid leak in S200 memory
- [SecLib\_RNG] 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() renamed 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

### Unsupported 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 Seed 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 firmware 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() has an important fix for reading the Timestamp in TSTMRO
  - 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\_INITRFSWITCHCONTROL Pins 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 the 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 HWPParameter.h
- [Platform]
  - Implement PLATFORM\_GetIeee802\_15\_4Addr() API in fwk\_platform\_ot.c - New gPlatformUseUniqueIdFor15\_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 functionality, 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 (dc/dc)
  - 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 trimming 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 oscillator 32k to 3
- [platform] Switch back to RNG for MAC Address 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 gBoard-UseUart0HwFlowControl, 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 re-generated

### 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

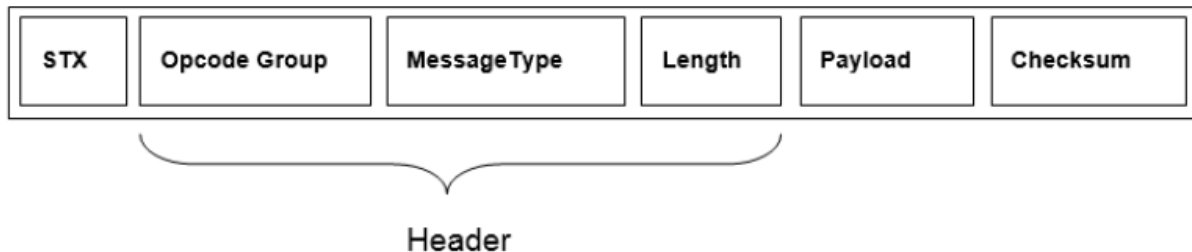
layers. In this setup, the user can run numerous commands to test the Black Box application services and interfaces.

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.

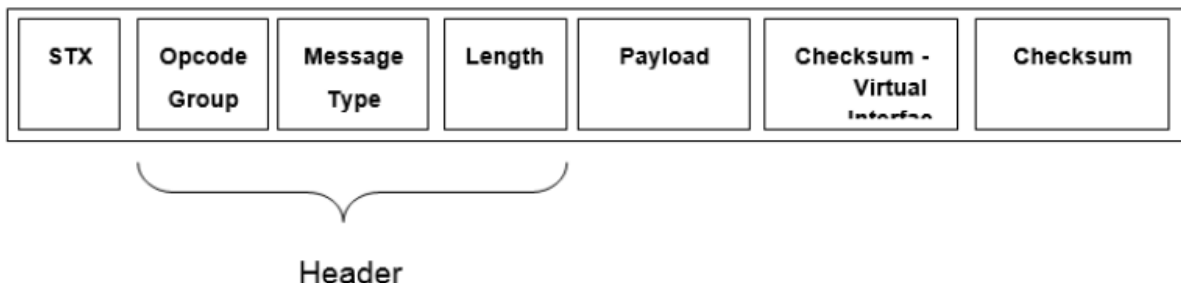
An entity which needs to be interfaced to the FSCI module can use the API to register opcodes to specific interfaces. After doing so, any packet coming from that interface with the same opcode triggers a callback execution. Two or more entities cannot register the same opcode on the same interface, but 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.<br>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.<br>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 */
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_transmitFormattedPacket(pClientPacket, interfaceId);
 return;
 }
 case 0x02:
 {
```

(continues on next page)

(continued from previous page)

```
/* Allocate 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_transmitFormattedPacket(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
 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 allotted 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) gHwParamsProdDataMainFlash2IfirMode\_c(1) : - PROD\_DATA are located in IFR1, but Main-Flash 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: gHwParamsProdDataPlacemen-tifirMode\_c) may be programmed.
- 3) gHwParamsProdDataIfirMode\_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 gUseProdInfoMain-Flash\_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 resources (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.

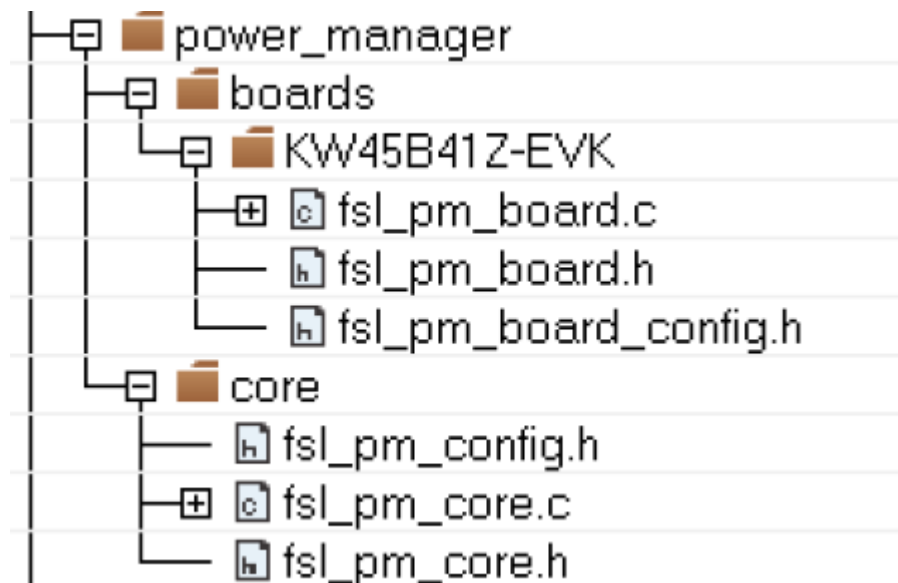
Other peripherals that require specific action on low power entry and restore on low power exit should be added to low power board files. For more details, refer to section Low power board file update

**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.

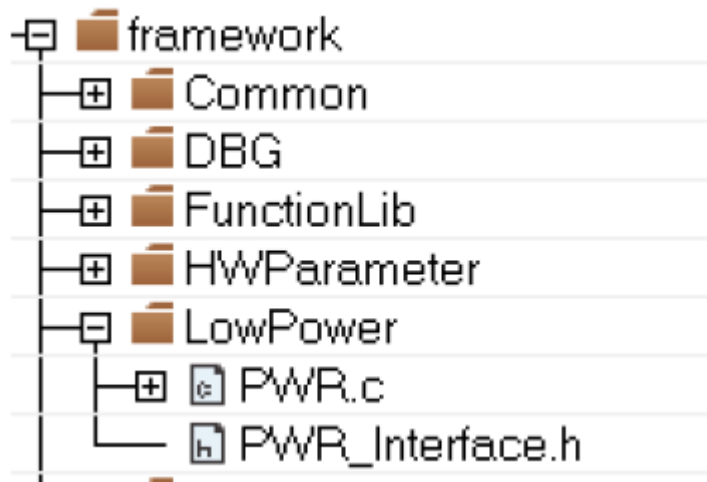
However, users can start directly from the application project and implement low power in it, by performing the steps described in the following sections.

**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.

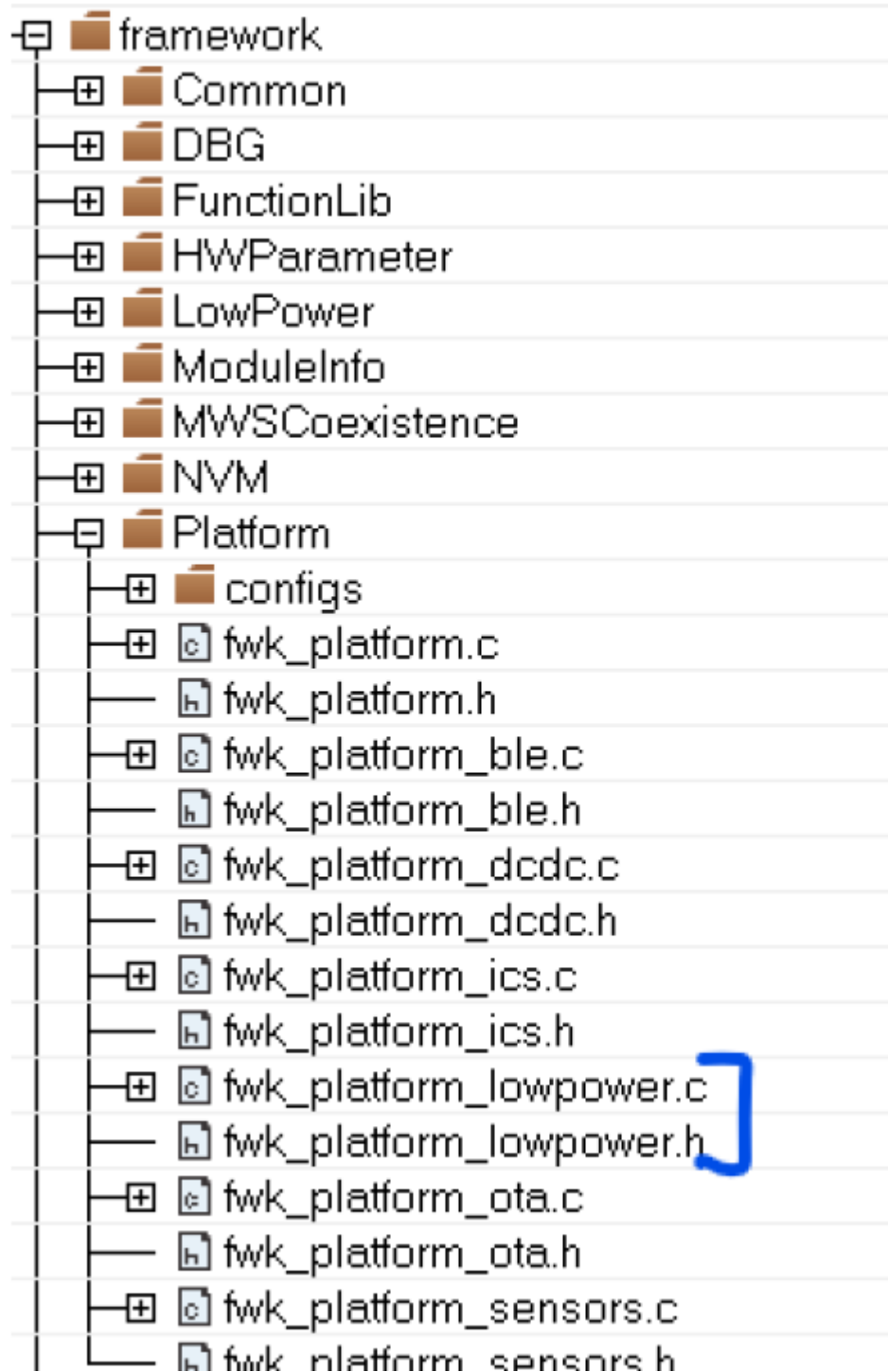
**2.1.2 - PWR connectivity framework module** `PWR.c` `PWR_Interface.h` shall be added to your application projects :



Optionally, in order to support SysTick less mode, `PWR_systicks.c` or `PWR_systicks_bm.c` could also be added.

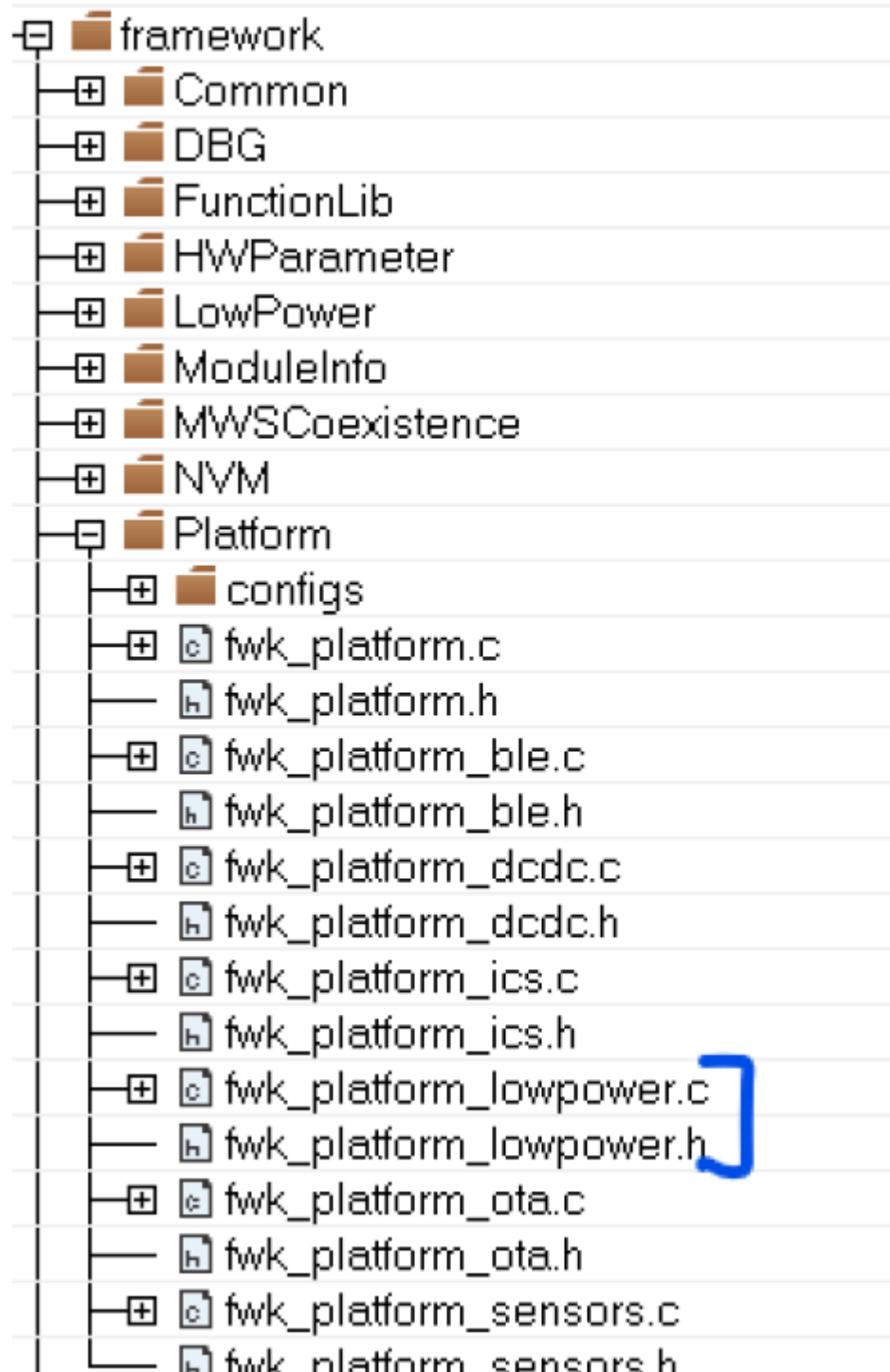
The include path to add is: `middleware/wireless/framework/LowPower`

**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 tickless 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 serial 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 call-back 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 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 aggressive 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 definition 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 written 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 `gApp-TaskWaitTimeout_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 Manager 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 suspended, 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 manual 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 save 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 take 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 than 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<br>AutoRestore |
| 0x1FFFF200 | 2         | 6        | 0x118F  | MirroredInRam                   |

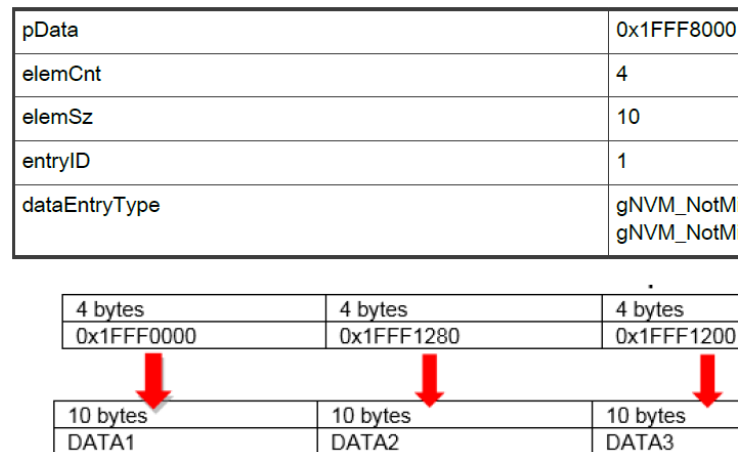
**NVM Table entry** As show above, A NVM table entry contains a generic pointer to a contiguous RAM data structure, the number of elements the structure contains, the size of a single element, a table entry ID, and an entry type.

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.



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  $0x1FFF8000 + 2 * \text{elemSz}$ . 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  $0x1FFF8000 + 2 * \text{sizeof(void*)}$ .

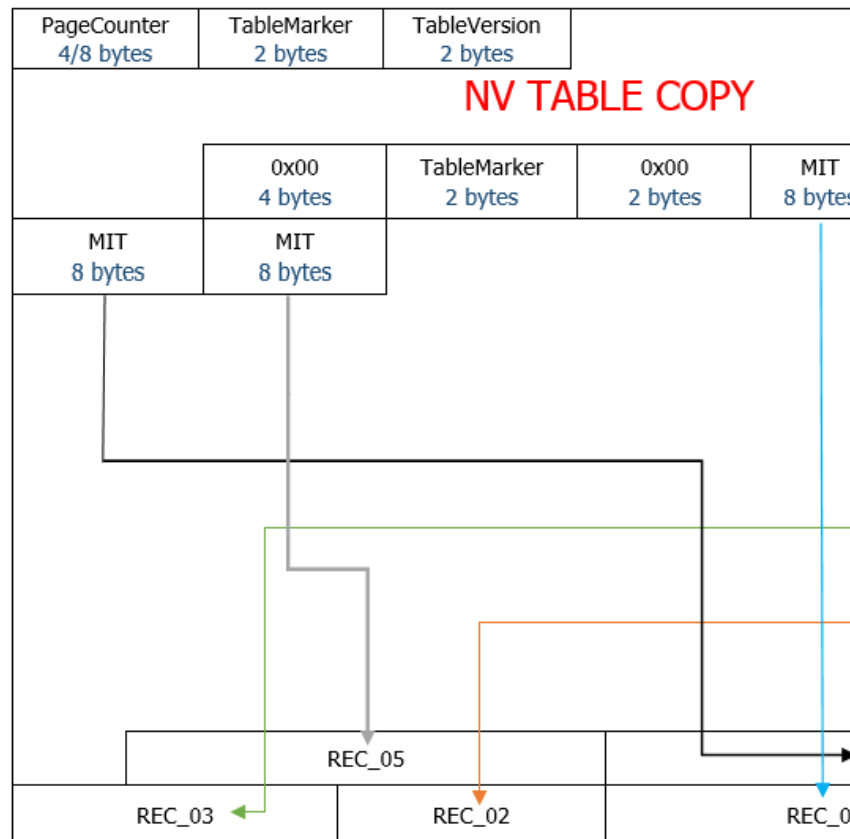
The page validity is guaranteed by the page counter. The page counter is a 32-bit value and is written at the beginning and at the end of the active page. The values need to be equal to consider the page a valid one. The value of the page counter is incremented after each page copy operation. A page 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

| entryId | entryType | elemCnt | elemSz  |
|---------|-----------|---------|---------|
| 2 bytes | 2 bytes   | 2 bytes | 2 bytes |

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:

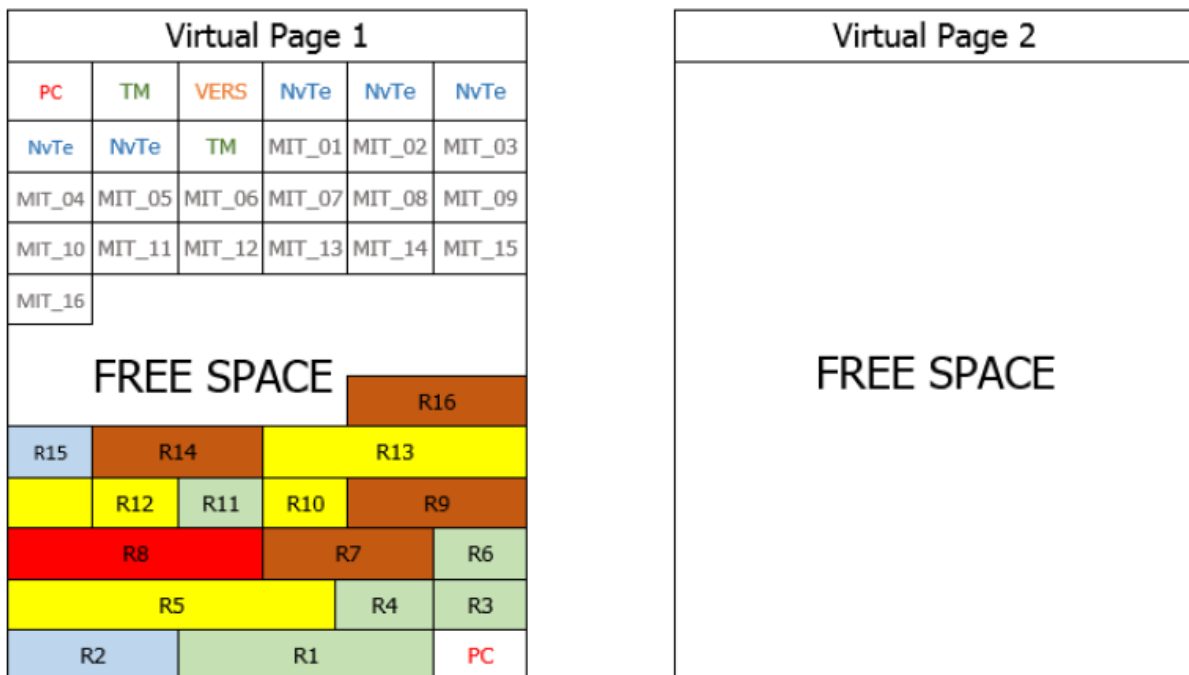
|            |                |                |                     |            |
|------------|----------------|----------------|---------------------|------------|
| <b>VSB</b> | <b>entryID</b> | <b>elemIdx</b> | <b>recordOffset</b> | <b>VEB</b> |
| 1 byte     | 2 bytes        | 2 bytes        | 2 bytes             |            |

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.

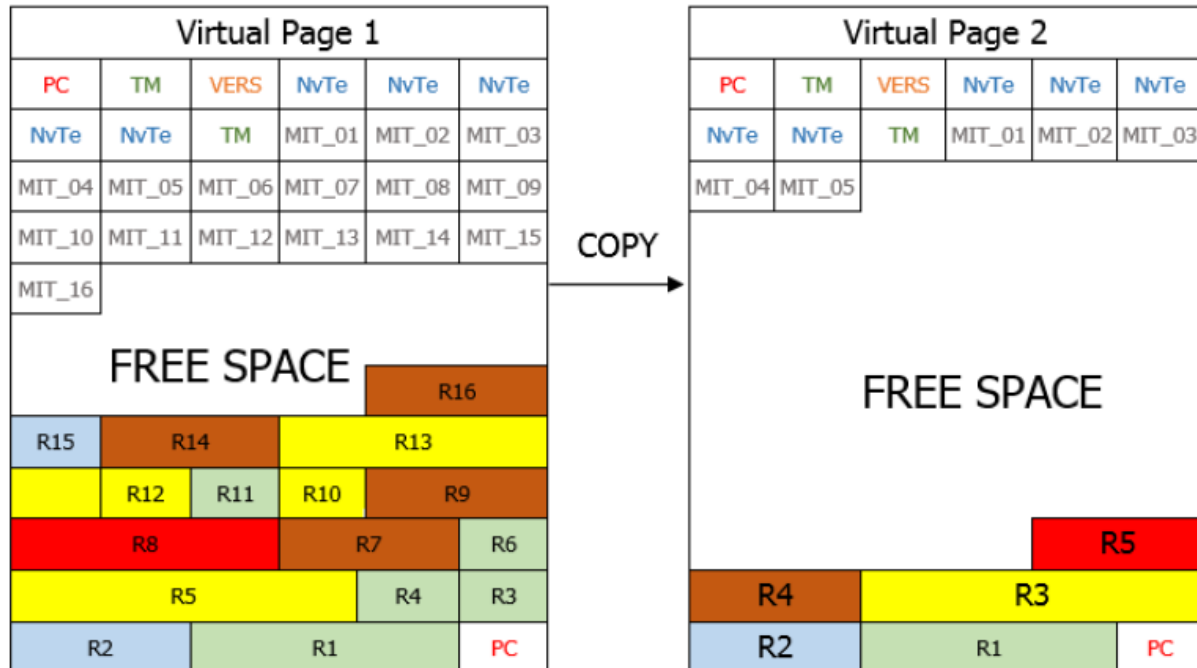


In this example, there are five datasets (one color for each dataset) with both 'full' and 'single' record types.

- 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 bus 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 upgradeIt 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

### 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 alternatively 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 subsystem. 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 from this hardware synchronously. In the case of an NBU that does not control the device's entropy source, that is owned by the Host, it requests 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. From the Idle thread, this flag is polled via the `RNG_IsReseedNeeded` API. If set the seed is regenerated and forwarded to the NBU via RPMSG.

`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.

**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.

## 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_ecdh.c | Mbedtls SecLib_mbedtls | nccl (part of SecLib.c) | Usage example             |
|-------------------------------------------------------------|-----|----------------------------------------------|-------------------------------|---------------|------------------------|-------------------------|---------------------------|
| AES_128                                                     |     | SecLib_aes.c                                 | x                             |               | x                      |                         |                           |
| AES_128_ECB                                                 |     |                                              | x                             |               | x                      |                         |                           |
| AES_128_CBC                                                 |     | x                                            | x                             |               | x                      |                         |                           |
| AES_128_CTR encryption                                      | en- | x                                            | x                             |               |                        |                         |                           |
| AES_128_OFB encryption                                      | En- | x                                            |                               |               |                        |                         |                           |
| AES_128_CMAC                                                |     | x                                            | x                             |               | x                      |                         | BLE connection, ieee 15.4 |
| AES_128_EAX                                                 |     | x                                            |                               |               |                        |                         |                           |
| AES_128_CCM                                                 |     | x                                            | x                             |               | x                      |                         | BLE, ieee 15.4            |
| SHA1                                                        |     | SecLib_sha.c                                 | x                             |               | x                      |                         |                           |
| SHA256                                                      |     | 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                      | 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                    |

## 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 preventing lowpower (only WFI) during its processing. Platform specific code can be find in `fwk_platform_sensors.c/h`.

#### 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 swich 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:  $\text{new\_estimation} = (\text{new\_measurement} + ((1 \ll n) - 1) * \text{last\_estimation}) \gg n$

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 FRO32K. 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.

## Wireless IEEE

### K32W1 / MCXW71 / MCXW72 Connectivity Test Application Getting Started User Guide (UG10204)

**Introduction** This document describes briefly how to build the Connectivity Test Application for the Arm Cortex based K32W1, MCXW71, and MCXW72 wireless microcontroller platforms provided by NXP.

**Building the binary file** This section describes the required steps for obtaining the binary file for usage with the board. The Connectivity Test Application is available in the package location below:

```
<sdk_folder_path>\boards\<board_name>\wireless_examples\ieee-802.15.4\connectivity_test\bm\iar
```

Where <board name> is:

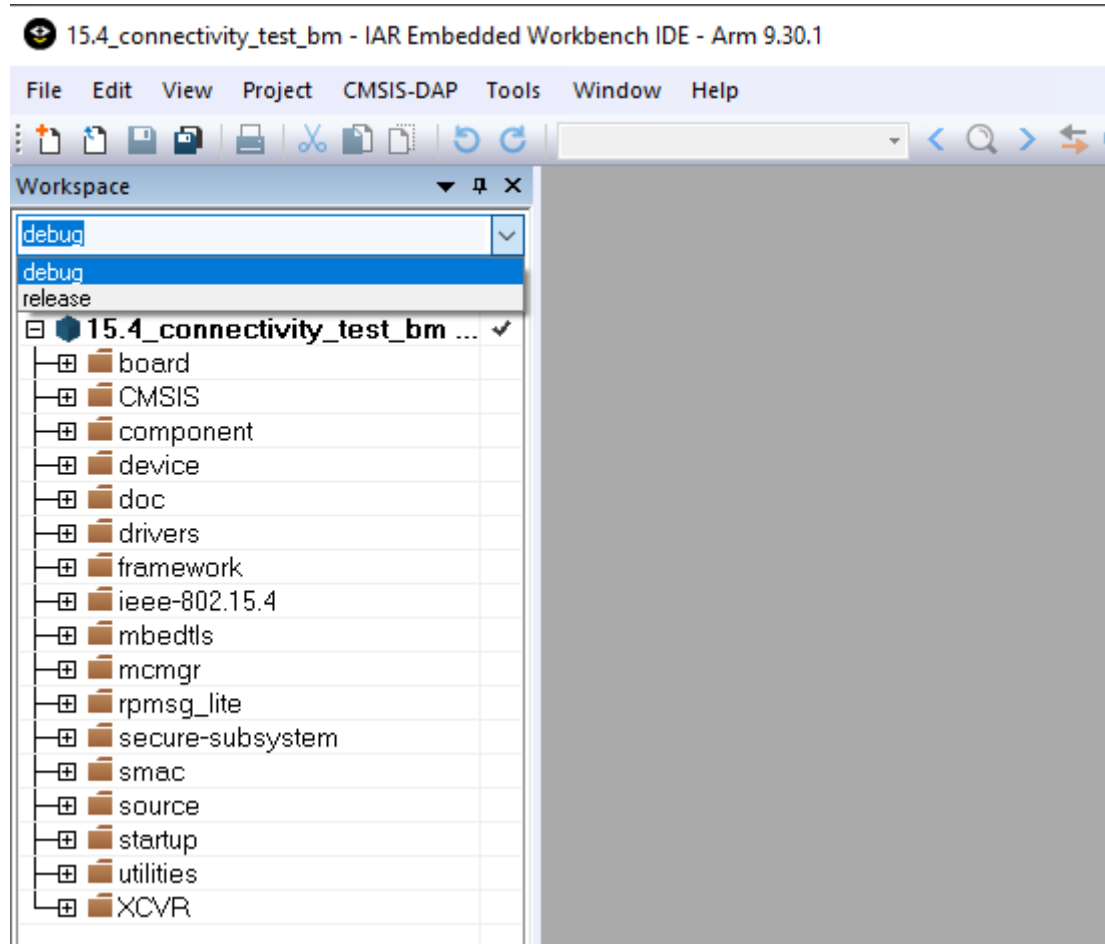
- k32w148evk for K32W1480-EVK
- frdm-mcxw71 for FRDM-MCXW71
- frdm-mcxw72 for FRDM-MCXW72
- mcxw71evk for MCX-W71-EVK
- mcxw72evk for MCX-W72-EVK

### Building and flashing the Connectivity Test application using IAR

1. **Step 1:** Navigate to the Connectivity Test Application location described above.
2. **Step 2:** Open the highlighted IAR IDE Workspace file (\*.eww file format:)

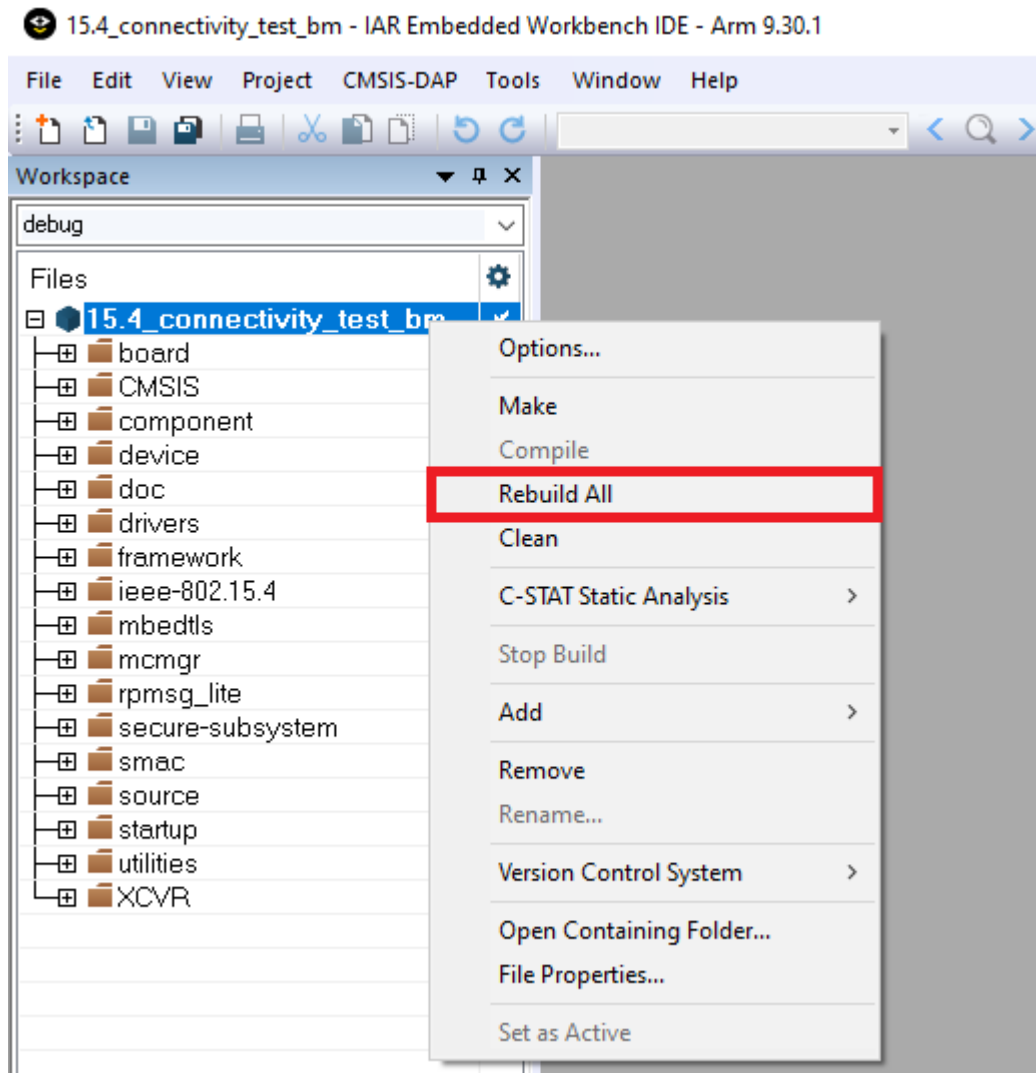
|                                                                                     |                               |                   |                   |
|-------------------------------------------------------------------------------------|-------------------------------|-------------------|-------------------|
|  | 15.4_connectivity_test_bm.ewd | 2/12/2023 1:58 PM | EWD File          |
|  | 15.4_connectivity_test_bm.ewp | 2/12/2023 1:58 PM | EWP File          |
|  | 15.4_connectivity_test_bm     | 2/12/2023 1:58 PM | IAR IDE Workspace |
|  | connectivity.icf              | 2/12/2023 1:58 PM | ICF File          |

3. **Step 3:** Select the desired configuration for the Connectivity Test Application project:

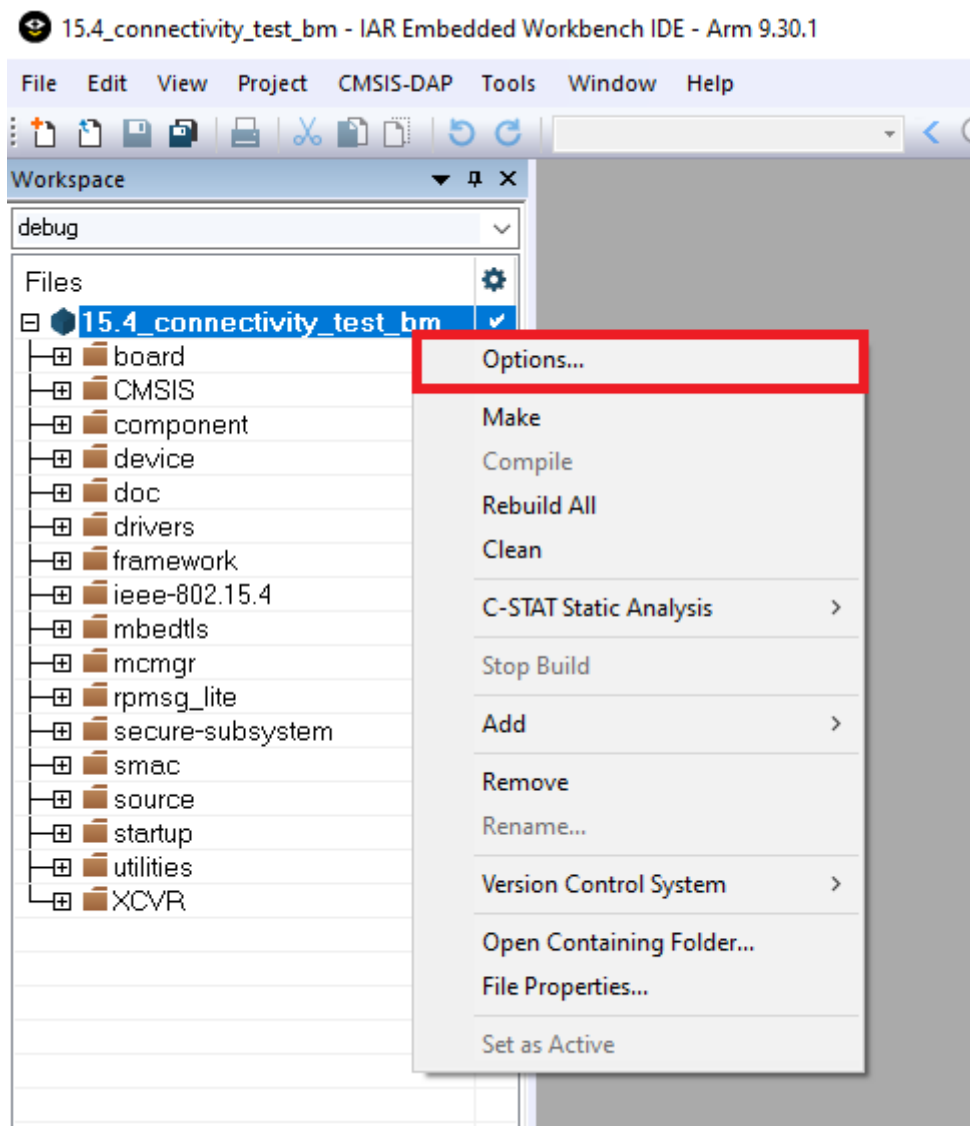


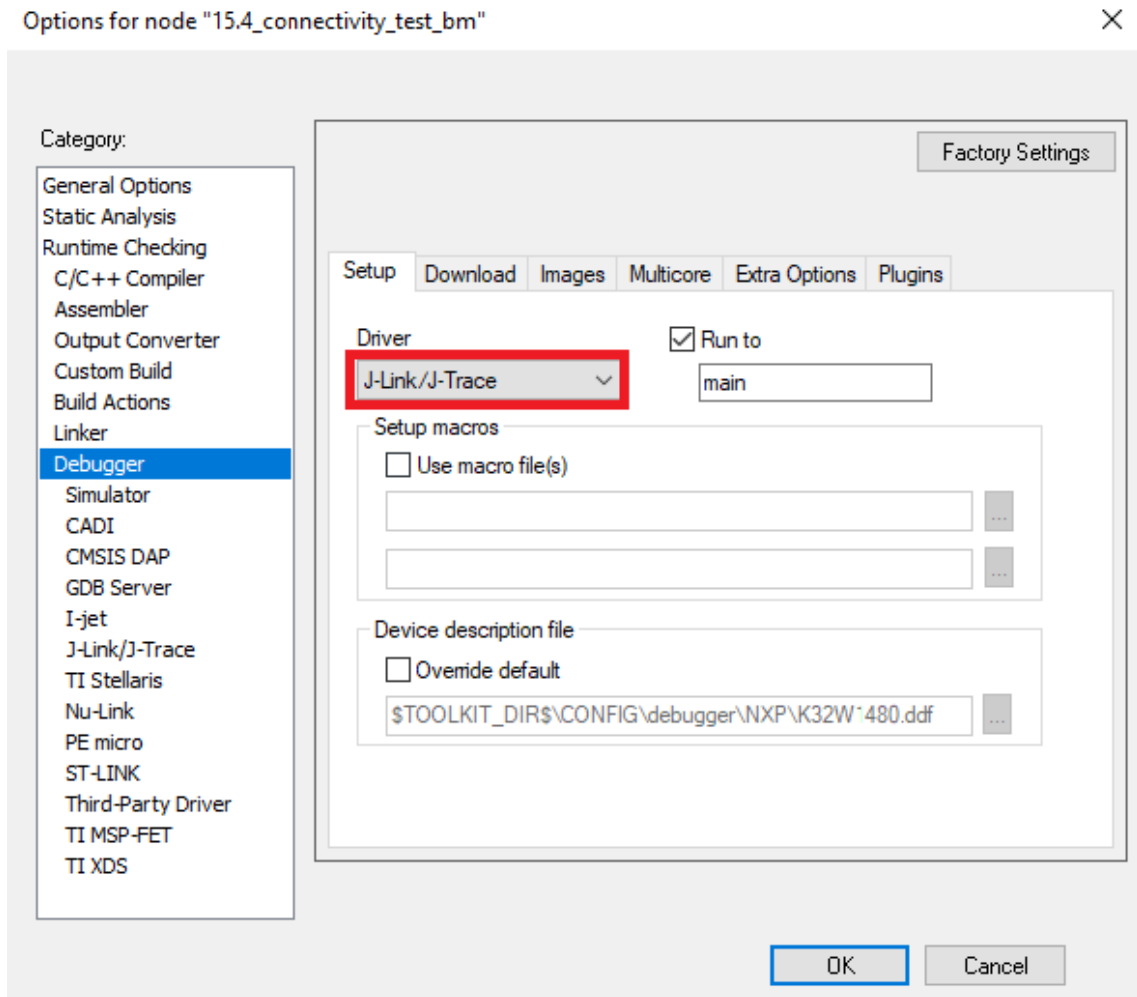
4. **Step 4:** Build the project.



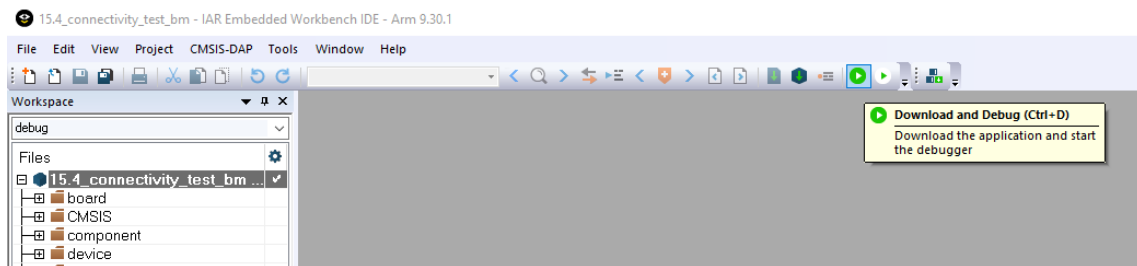


5. **Step 5:** Make the appropriate debugger settings in the project options window:





6. **Step 6:** Click the “**Download and Debug**” button to flash the executable onto the board.

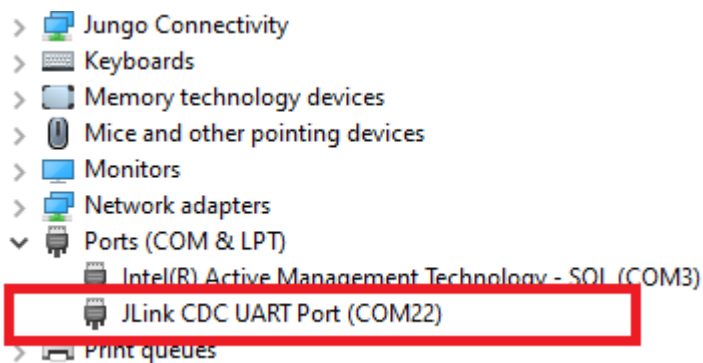


**Running the Connectivity Test application** This section demonstrates the basic steps to run a demo application.

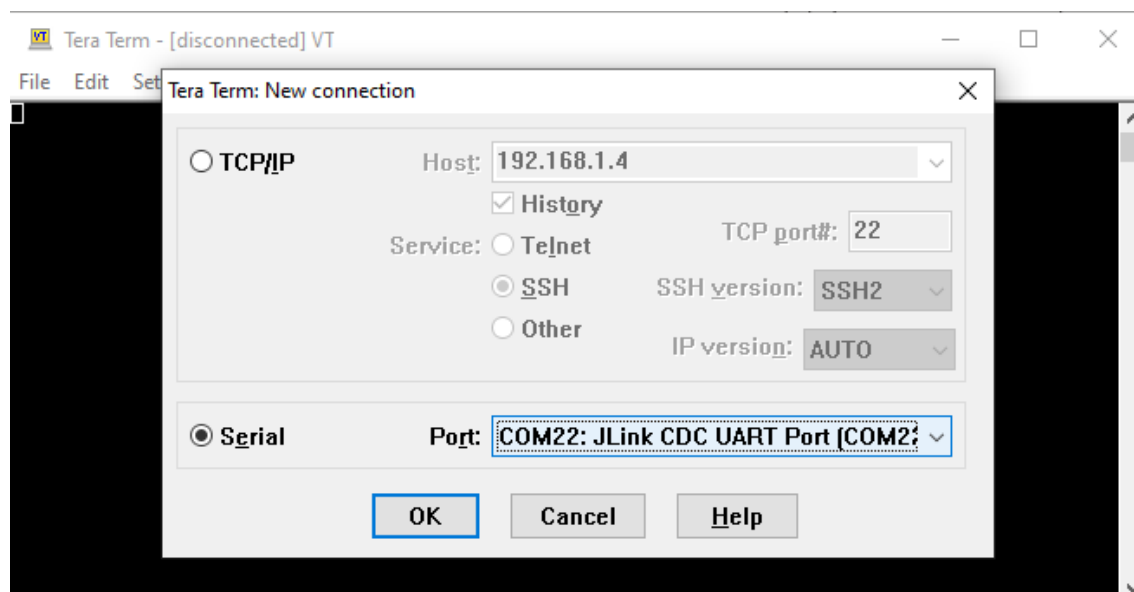
**Note:** For detailed information about the Connectivity Test Application, refer to the “*Connectivity Test Application Command Line Interface User Guide*” included in the package (CTA-CLIUG\_Rev0.pdf).

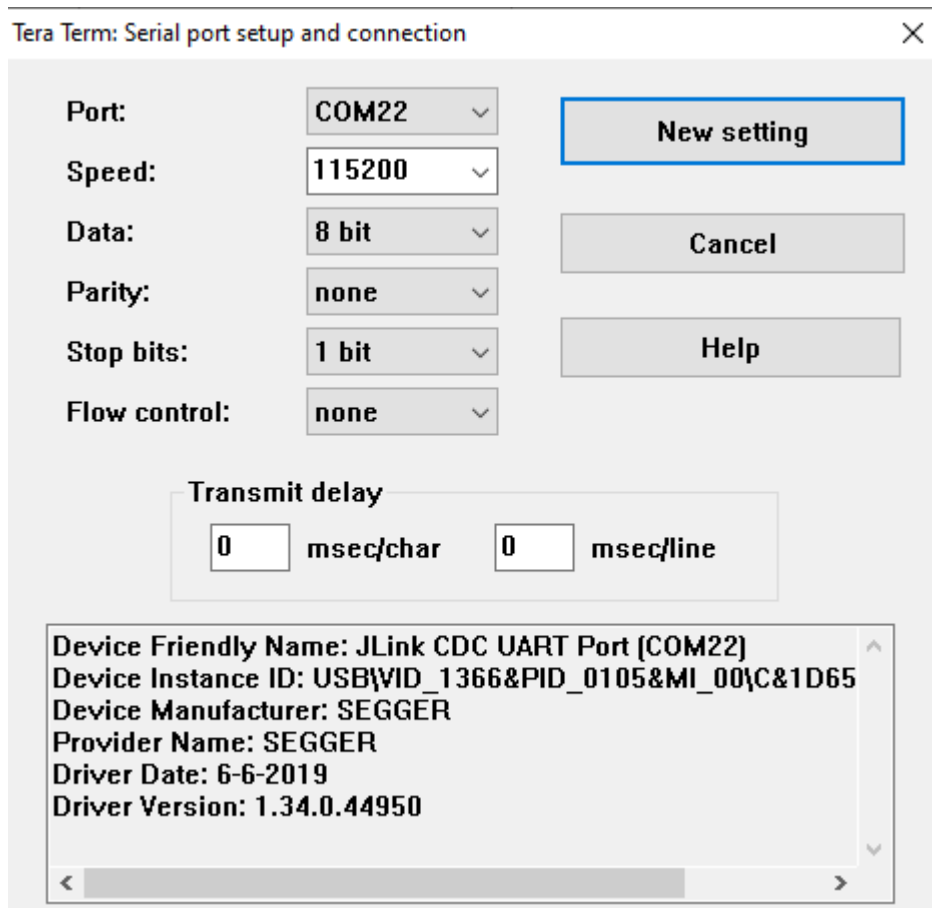
In order to connect to the board, the Connectivity Test application requires a serial terminal program. This example uses Tera Term for this purpose.

1. **Step 1:** Load the application on the board using IAR Embedded Workbench for Arm by clicking “**Download and Debug**”.
2. **Step 2:** After loading the application, check “**Device Manager**” to get the serial port number. This should appear with the prefix “**JLink**”.

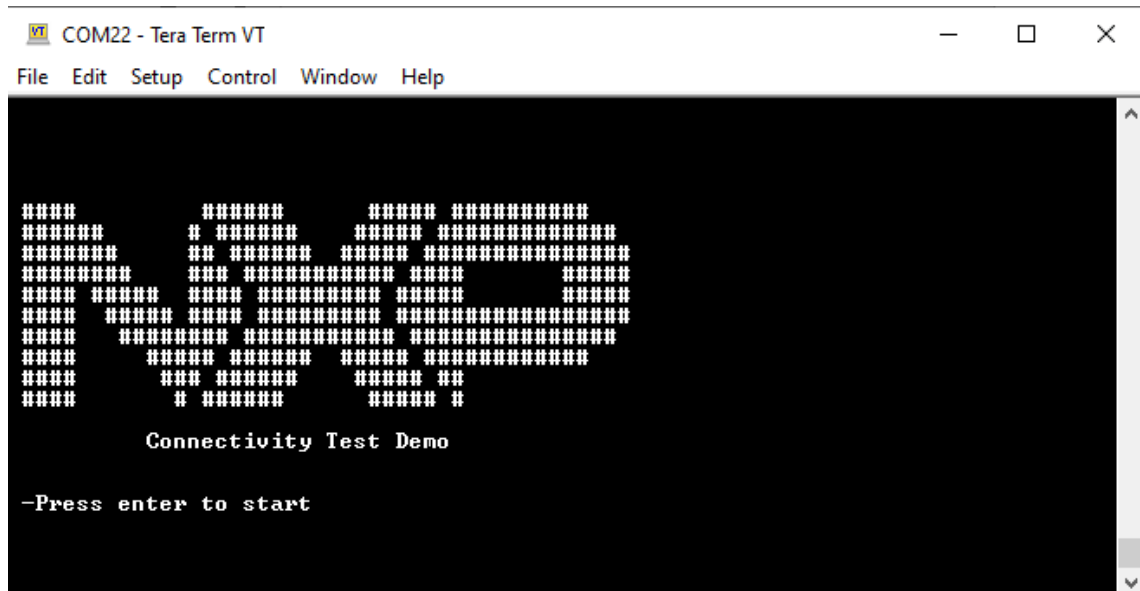


3. **Step 3:** Using the port numbers specified in Device Manager, open a Tera Term instance and connect to the device using the 115200 baud rate. To change the baud rate of the terminal go to the “**Setup-> Serial Port**” menu.





4. **Step 4:** Start the application by pressing the “Enter” key. Any other key displays the logo screen again.



5. Follow the on-screen instructions to run each test. If a test needs a second platform, follow the steps above to set it up.

**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 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 the revisions to this document.

| Document ID       | Release date     | Description                                 |
|-------------------|------------------|---------------------------------------------|
| UG10204 v.1.0     | 24 January 2025  | Added support for MCXW71 and MCXW72 devices |
| K32W1CTAGSG v.0.0 | 27 February 2023 | Initial release for K32W1 platform          |

## CTACLIUG

**Introduction** This document describes the Command Line Interface (CLI) options provided with the **812.15.4 Connectivity Test application**. The application provides tests for measuring transmission between two or more devices. This document also provides the test description, configuration, setup, and steps to perform the test using the application.

**Connectivity Test application setup** The figure below describes the Connectivity Test Application possible configurations.

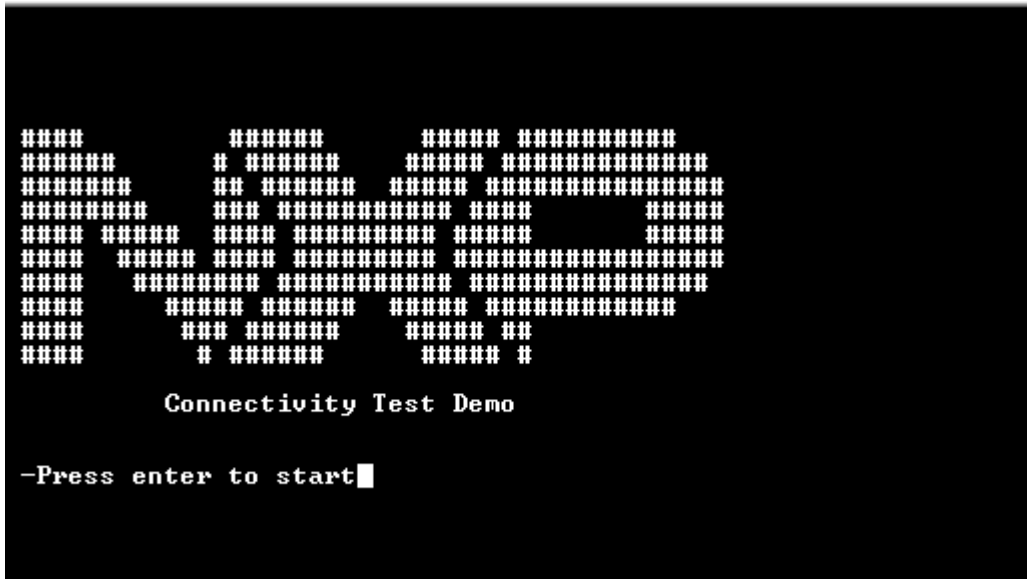
### Test bed description

- Running the Connectivity Test Demo Application requires a PC and two devices.
- Connectivity Test Application CLI is accessed using the serial port of the devices.
- Some features of the Connectivity Test Application like **Packet Error Rate** or **Range Tests** require the usage of two devices.
- For capturing over the air traffic, an RF sniffer can be used.
- For tests where the output power or the modulated output power of the radio should be measured, a spectrum analyzer should be used.

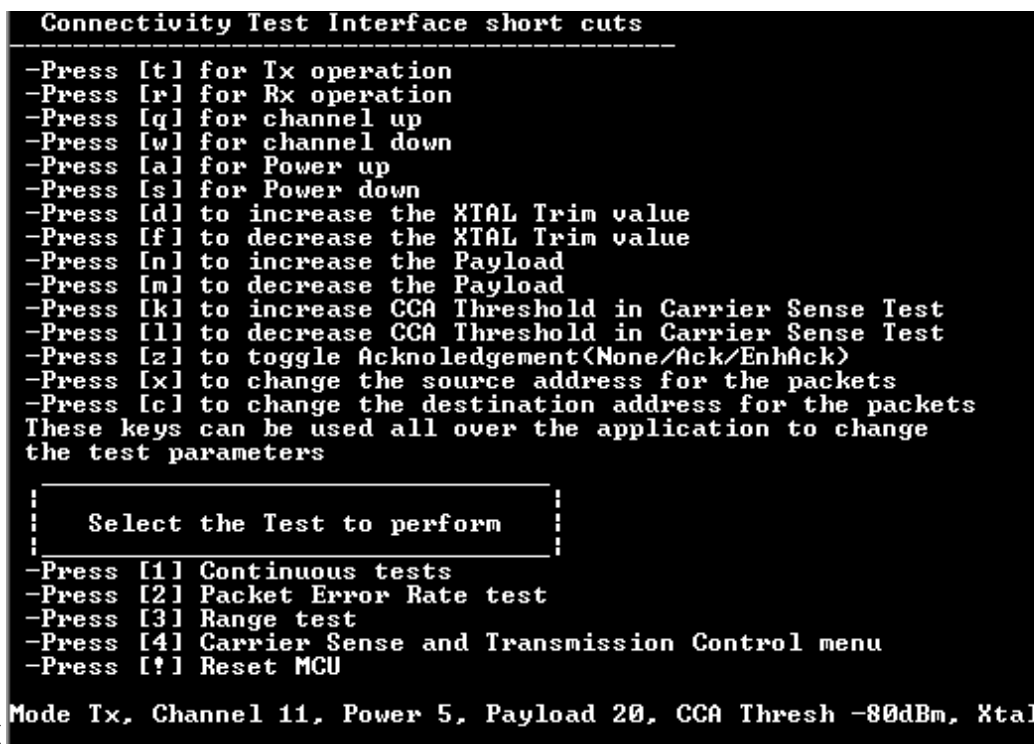
**Parent topic:**[Connectivity Test application setup](#)

## Connectivity Test CLI

**CLI initialization and description** The figure below shows how the Command Line Interface (CLI) menu of the **812.15.4 Connectivity Test application** appears at start-up.



Press “**Enter**” on the message prompt. The full CLI menu is displayed as shown in the figure below.



### Connectivity Test Demo CLI

The CLI is divided into three main categories:

1. **Shortcuts:** This category provides configuration options that can be used in the tests performed. Shortcuts can be used within any accessed test menu to change the test parameters.
2. **Tests:** This category provides the test that can be performed. Each test category numbered from 1 to 4 contains additional options.
3. **Device state:** This category indicates the current state of the device. The device state can be modified according to user preference by using the shortcut keys available by the CLI.



Device state is displayed within any test menu and is refreshed every time a shortcut command is issued. Device state is not shown when tests are in progress.

Parent topic: [Connectivity Test CLI](#)

**CLI shortcut command description** Commands from this section are used for configuring the device. All the shortcut commands modifications, except **\*\*[x]\*\*** and **[c]**, are displayed in the device state.

Shortcut commands cover the following configuration settings:

1. **Device operation mode:**

- **[t]**: configures the device in **transmitter** mode
- **[r]**: configures the device in **receiver** mode

2. **Channel settings:**

- Channel number range between [11-26], default value = 11
- **[q]:increments** channel number
- **[w]: decrements** channel number

3. **Output power settings:**

- Output power range between [0-32], default value = 5
- **[a]: increments** output power
- **[s]: decrements** output power

4. **Crystal trim value settings:**

- XTAL trim range between [0-127], default value = 0
- **[d]: increases** XTAL trim value
- **[f]: decreases** XTAL trim value

5. **Data payload length settings:**

- **[n]: increases** the data payload length
- **[m]: decreases** the data payload length

6. **CCA threshold settings:**

- CCA Threshold range between [0-110], default value=80
- **[k]: increases** CCA Threshold in Carrier Sense Test
- **[l]: decreases** CCA Threshold in Carrier Sense Test

7. **Packet acknowledgment settings:**

- Default value = NoAck.
- **[z]**: sets Acknowledgement requirement (NoAck/Ack/EnhancedAck).

8. **Packet source/address settings:**

- Source address default value = 0xBEAD
- Destination address default value = 0xFFFF
- **[x]**: sets **source** address for packets
- **[c]**: sets **destination** address for packets

Parent topic: [Connectivity Test CLI](#)

**CLI test description** Four major test categories are exposed through Connectivity Test Application:

1. **Continuous tests**
2. **Packet Error Rate Test**
3. **Range Test**
4. **Carrier Sense and Transmission Control**

**Continuous Test menu** The Continuous Test menu displays the following modes of operation:

1. **Idle mode:** This mode sets the transceiver into Idle mode.
2. **Burst PRBS transmission using packet mode:** This mode verifies the transceiver operational state by transmitting over the air a Pseudo Random Binary Sequence payload of 65 bytes.
3. **Continuous modulated transmission:** This mode generates and loads a PRBS9 pattern into transceiver's TX buffer and is used to verify the modulated output power of the transceiver.
4. **Continuous unmodulated transmission:** This mode sets the transceiver to continuously transmit an unmodulated carrier (CW) and is used to measure the output power of the transceiver.
5. **Continuous reception:** This mode sets the transceiver into Receive sequence.
6. **Continuous energy detect:** This mode sets the transceiver to perform the Energy Detection operation. It returns the maximal energy on the specified channel.
7. **Continuous scan:** This mode returns values of Energy Detect operation performed on all RF channels.
8. **Continuous CCA:** This mode performs Continuous Clear Channel Assessment (CCA) procedure and returns whether the specified channel is Busy or Idle.

**Parent topic:**CLI test description

**Packet Error Rate (PER) menu** The Packet Error Rate (PER) test mode is used to measure the sensitivity of the radio based on packet reception. The content of this menu is strictly related to the operation mode of the device.

PER test can be performed in three different modes:

1. Transmission *without expected acknowledgement*.
2. Transmission *with expected acknowledgement*.
3. Transmission *with expected enhanced acknowledgement*.

**Steps to perform a PER test** The steps to perform a PER test procedure are as follows:

1. Set the source and destination address of the transmitter device. For example, source=0xaaaa, destination=0xbbbb. (Any other address can be chosen instead.) See the figure below.

```
PER Test: Set source and destination addresses on TX side
Change src address <AAAA> [0-9 a-z symbols] >aaaa
Mode Tx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >
Change dst address <BBBB> [0-9 a-z symbols] >bbbb
Mode Tx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >■
```

2. Set the source and destination address of the receiver device. For example, source=0xbbbb, destination=0xaaaa (Any other addresses can be chosen instead). See the figure below.

**PER Test: Set destination and destination addresses on RX side**

```
Change src address <BBBB> [0-9 a-z symbols] >bbbb
Mode Rx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >
Change dst address <AAAA> [0-9 a-z symbols] >aaaa
Mode Rx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >
```

3. Set the receiver device to run.
4. Set the transmitter device to run.

**Parent topic:**Packet Error Rate (PER) menu

**Packet error rate when the device is set as transceiver** When the device is set to Transmitter mode, it prompts the user to indicate the following:

- The number of packets that should be transmitted during the test.
- The transmission interval between two consecutive packets.

See the figure below. **Configuring PER for TX device with no ACK**

```
PER Tx Test Menu

Choose the amount of packets to send:
[0] - 1 Packet [1] - 25 Packets
[2] - 100 Packets [3] - 500 Packets
[4] - 1000 Packets [5] - 2000 Packets
[6] - 5000 Packets [7] - 10000 Packets
[8] - 65535 Packets

-Press [p] Previous Menu
Mode Tx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >
Please type TX interval in milliseconds < > 5ms > and press [ENTER]
█
```

If the acknowledgment setting has not been previously defined, users can decide whether acknowledgment should be present or not.

The type of acknowledgments that can be used by issuing shortcut commands can also be set. See the figure below. **Configuring PER for TX device with ACK**

```
PER Tx Test Menu

Choose the amount of packets to send:
[0] - 1 Packet [1] - 25 Packets
[2] - 100 Packets [3] - 500 Packets
[4] - 1000 Packets [5] - 2000 Packets
[6] - 5000 Packets [7] - 10000 Packets
[8] - 65535 Packets

-Press [p] Previous Menu
Mode Tx[ack], Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >
Please type TX interval in milliseconds < > 5ms > and press [ENTER]
```

See the figure below. **Configuring PER for TX device with Enhanced ACK**

```

PER Tx Test Menu

Choose the amount of packets to send:
[0] - 1 Packet [1] - 25 Packets
[2] - 100 Packets [3] - 500 Packets
[4] - 1000 Packets [5] - 2000 Packets
[6] - 5000 Packets [7] - 10000 Packets
[8] - 65535 Packets

Press [p] Previous Menu
Mode Tx[enhAck], Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >
Please type TX interval in milliseconds < > 5ms > and press [ENTER]

```

With everything being set, the device starts transmitting the packets over the air as shown in the figure below. **Running PER on TX side**

```

Please type TX interval in milliseconds < > 5ms > and press [ENTER]
20

Running PER Tx, Sending 25 Packets
Packet 1
Packet 2
Packet 3
Packet 4
Packet 5
Packet 6
Packet 7
Packet 8
Packet 9
Packet 10
Packet 11
Packet 12
Packet 13
Packet 14
Packet 15
Packet 16
Packet 17
Packet 18
Packet 19
Packet 20
Packet 21
Packet 22
Packet 23
Packet 24
Packet 25
PER Tx DONE

```

The packet or sequence of packets that have not been acknowledged are indicated on the transmitter CLI. This is applicable for transmissions with acknowledgement or enhanced acknowledgement set, if the receiver device does not send an acknowledgment as required. See the figure below.

```

Running PER Tx, Sending 25 Packets
Packet 1 <no-ack>
Packet 2 <no-ack>
Packet 3 <no-ack>
Packet 4 <no-ack>
Packet 5 <no-ack>
Packet 6 <no-ack>

```

Example of not acknowledged packets

Parent topic:Packet Error Rate (PER) menu

**Packet error rate menu when the device is set as receiver** When the device is set to receiver mode, the user is only asked to start or stop the receiving sequence as shown in the figure below.

## PER menu interface for RX device

## PER Rx Test Menu

-Press [space bar] to start/stop Receiving Packets  
 -Press [p] Previous Menu

Mode Rx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm

After the receiving sequence begins, incoming packages with destination address set to device's destination address are shown in the figure below. **Running PER on RX side**

```
PER Test Rx Running
Packet 1. Packet index: 1. Rssi during RX: -97
Packet 2. Packet index: 2. Rssi during RX: -97
Packet 3. Packet index: 3. Rssi during RX: -97
Packet 4. Packet index: 4. Rssi during RX: -97
Packet 5. Packet index: 5. Rssi during RX: -97
Packet 6. Packet index: 6. Rssi during RX: -97
Packet 7. Packet index: 7. Rssi during RX: -97
Packet 8. Packet index: 8. Rssi during RX: -97
Packet 9. Packet index: 9. Rssi during RX: -97
Packet 10. Packet index: 10. Rssi during RX: -97
Packet 11. Packet index: 11. Rssi during RX: -97
Packet 12. Packet index: 12. Rssi during RX: -97
Packet 13. Packet index: 13. Rssi during RX: -97
Packet 14. Packet index: 14. Rssi during RX: -97
Packet 15. Packet index: 15. Rssi during RX: -97
Packet 16. Packet index: 16. Rssi during RX: -97
Packet 17. Packet index: 17. Rssi during RX: -97
Packet 18. Packet index: 18. Rssi during RX: -97
Packet 19. Packet index: 19. Rssi during RX: -97
Packet 20. Packet index: 20. Rssi during RX: -97
Packet 21. Packet index: 21. Rssi during RX: -97
Packet 22. Packet index: 22. Rssi during RX: -97
Packet 23. Packet index: 23. Rssi during RX: -97
Packet 24. Packet index: 24. Rssi during RX: -97
Packet 25. Packet index: 25. Rssi during RX: -97
Average Rssi during PER: -97 dBm
PER Test Finished
Received 25 of 25 packets transmitted
```

**Parent topic:**Packet Error Rate (PER) menu

**Parent topic:**CLI test description

**Range Test menu** This is a simple and quick test in which data packets are exchanged between two devices.

- On the transmitter side, the device initiates the data transfer and expects to receive a packet containing the same data payload. An average RSSI calculation is performed based on the incoming packet's RSSI and the number of sent packets. If the receiving packet does not contain the same data payload, RSSI calculation is not performed, and a 'packet dropped' message is shown in CLI.
- On the receiver side, the device sends back the message to originator.

Range tests can be performed in three different modes:

1. Transmission without expected Acknowledgment.
2. Transmission with expected Acknowledgment.
3. Transmission with expected Enhanced Acknowledgment

**Steps to perform a Range Test procedure** The steps to perform a Range Test procedure are as follows:

1. Set the source and destination address of the transmitter device: source=0xaaaa, destination=0xbbbb. (Any other addresses can be chosen). See the figure below. **Range Test: Set the source and destination addresses on TX side**

```
Change src address <AAAA> [0-9 a-z symbols] >aaaa
Mode Tx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >
Change dst address <BBBB> [0-9 a-z symbols] >bbbb
Mode Tx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >
```

2. Set the source and destination address of the receiver device.

For example, source=0xbbbb, destination=0xaaaa. (Any other addresses can be chosen). See the figure below.

**Range Test: Set the source and destination addresses on RX side**

```
Change src address <BBBB> [0-9 a-z symbols] >bbbb
Mode Rx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >
Change dst address <AAAA> [0-9 a-z symbols] >aaaa
Mode Rx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >
```

3. Set the receiver device to run as shown in the figure below. **Range Test: Set the RX device**

```

Range Rx Test Menu
-Press [space bar] to start/stop Receiving Packets
-Press [p] Previous Menu
Mode Rx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >
Range Test Rx Running
```

to run

4. Set the transmitter device to run.

TX device CLI messages are shown in the figures below. **Range Test: TX device CLI mes-**

```

Range Tx Test Menu
-Press [space bar] to start/stop Transmitting Packets
-Press [p] Previous Menu
Mode Tx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >
```

sages

```

Range Test Tx Running
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
Range Test Tx Stopped
Average RSSI -97 dBm
Packets dropped 0
```

TX device CLI messages

5. Receiver device CLI messages are as shown in the figure below. **Range Test: RX device CLI messages**

```
Range Test Rx Running
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
RSSI = -97 dBm
Range Test Rx Stopped
Average RSSI -97 dBm
```

Parent topic:Range Test menu

Parent topic:CLI test description

## Carrier Sense and Transmission Control Select menu

**Carrier Sense Test with unmodulated input signal** This test performs a “manual” Clear Channel Assessment procedure. First, an energy request procedure is done to obtain the RSSI on the specified channel. Then, the obtained RSSI is compared against CCA threshold value, which can be manually set by issuing shortcut commands **\*\*[l]\*\*** and **[k]**.

If the obtained RSSI value for the specified channel is greater than the CCA threshold value, then the channel is considered busy, no transmission is issued on that channel and another energy detect request is started. This is shown in the below figure. **Carrier Sense Test when RSSI is greater than CCA Threshold**

```
Radio Carrier Sense and Transmission Control Select Menu
-Press [1] Carrier Sense Test with un-modulation input signal
-Press [2] Transmission Control Test
-Press [p] Previous Menu

Mode Tx, Channel 11, Power 32, Payload 20, CCA Thresh -102dBm, XtalTrim 0 >
Press [SPACE] to begin/interrupt test
Press [p] to return to previous menu

Sampling done. RSSI value: -101dBm
Sampling done. RSSI value: -99dBm
Sampling done. RSSI value: -102dBm
Sampling done. RSSI value: -102dBm
Sampling done. RSSI value: -101dBm
Sampling done. RSSI value: -91dBm
Sampling done. RSSI value: -101dBm
Sampling done. RSSI value: -101dBm
Sampling done. RSSI value: -103dBm
Transmission Performed
```

However, if the obtained RSSI value for the specified channel is lower than the CCA Threshold value, then the channel is considered as Idle. It indicates that a transmission has occurred and therefore, the test ends. This is shown in the below figure. **Carrier Sense Test when RSSI is lower than CCA Threshold**



```

: Radio Carrier Sense and Transmission Control Select Menu :
:
-Press [1] Carrier Sense Test with un-modulation input signal
-Press [2] Transmission Control Test
-Press [p] Previous Menu

Mode Tx, Channel 11, Power 32, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >

Press [SPACE] to begin/interrupt test
Press [p] to return to previous menu

 Sampling done. RSSI value: -102dBm
Transmission Performed

```

**Parent topic:**Carrier Sense and Transmission Control Select menu

**Transmission Control test** The Transmission Control test performs data transmissions over the air using user-specified options.

The data payload is composed from a predefined data buffer that copies as many bytes as the ‘device state payload data length setting’ indicates.

Before any transmission, the test performs an energy detect procedure to obtain the RSSI value of the specified channel. This value is displayed on the side of a message that indicates a message has been sent over the air.

**Steps to perform a Transmission Control Test** The figure below illustrates the Transmission Control Test menu. **Transmission Control Test menu**

```

: Tr Ctrl Test Menu :
:
Choose the amount of packets to send:
[0] - 1 Packet [1] - 25 Packets
[2] - 100 Packets [3] - 500 Packets
[4] - 1000 Packets [5] - 2000 Packets
[6] - 5000 Packets [7] - 10000 Packets
[8] - 65535 Packets

-Press [p] Previous Menu

Mode Tx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >

Please type InterPacket delay in milliseconds and press [ENTER]
<During test, exit by pressing [SPACE]>

```

The process to conduct the Transmission Control Test is as follows:

1. Select the number of packets to be sent.
2. Specify the inter-packet delay. (Transmission delay between two consecutive packets).

The figure shows this process. **Transmission Control Test execution**

```
Please type InterPacket delay in miliseconds and press [ENTER]
(During test, exit by pressing [SPACE])

10
 Packet number: 1; RSSI value: -104 dBm
 Packet number: 2; RSSI value: -106 dBm
 Packet number: 3; RSSI value: -104 dBm
 Packet number: 4; RSSI value: -104 dBm
 Packet number: 5; RSSI value: -78 dBm
 Packet number: 6; RSSI value: -98 dBm
```

This test also calculates an approximation of time needed to perform the Overhead + Transmission + Energy Detect procedure.

If the inter-packet delay has a value lower than the estimation previously calculated, a corresponding message is shown, and the test finishes the execution as shown in the figure below.

**Transmission Control Test – Inter-Packet delay error**

```

Tr Ctrl Test Menu

Choose the amount of packets to send:
[0] - 1 Packet [1] - 25 Packets
[2] - 100 Packets [3] - 500 Packets
[4] - 1000 Packets [5] - 2000 Packets
[6] - 5000 Packets [7] - 10000 Packets
[8] - 65535 Packets

Press [p] Previous Menu

Mode Tx, Channel 11, Power 5, Payload 20, CCA Thresh -80dBm, XtalTrim 0 >

Please type InterPacket delay in milliseconds and press [ENTER]
(During test, exit by pressing [SPACE])

3
Packet number: 1; RSSI value: -99 dBm
Overhead + Transmission + ED = ~4ms
Interpacket delay too small (Press [ENTER] to continue)

```

**Parent topic:**Transmission Control test

**Parent topic:**Carrier Sense and Transmission Control Select menu

**Parent topic:**CLI test description

**Parent topic:**[Connectivity Test CLI](#)

**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 the revisions to this document.

| Revision number | Date          | Substantive changes            |
|-----------------|---------------|--------------------------------|
| 1               | 25 March 2025 | Aligned for Sphinx environment |
| 0               | 9 Feb 2023    | Initial release                |



# Chapter 2

## RTOS

### 2.1 FreeRTOS

#### 2.1.1 FreeRTOS kernel

Open source RTOS kernel for small devices.

**FreeRTOS kernel for MCUXpresso SDK Readme**

**FreeRTOS kernel for MCUXpresso SDK**

**Overview** The purpose of this document is to describes the [FreeRTOS kernel repo](#) integration into the [NXP MCUXpresso Software Development Kit: mcuxsdk](#). MCUXpresso SDK provides a comprehensive development solutions designed to optimize, ease, and help accelerate embedded system development of applications based on MCUs from NXP. This project involves the FreeRTOS kernel repo fork with:

- cmake and Kconfig support to allow the configuration and build in MCUXpresso SDK ecosystem
- FreeRTOS OS additions, such as [FreeRTOS driver wrappers](#), RTOS ready FatFs file system, and the implementation of FreeRTOS tickless mode

The history of changes in FreeRTOS kernel repo for MCUXpresso SDK are summarized in [CHANGELOG\\_mcuxsdk.md](#) file.

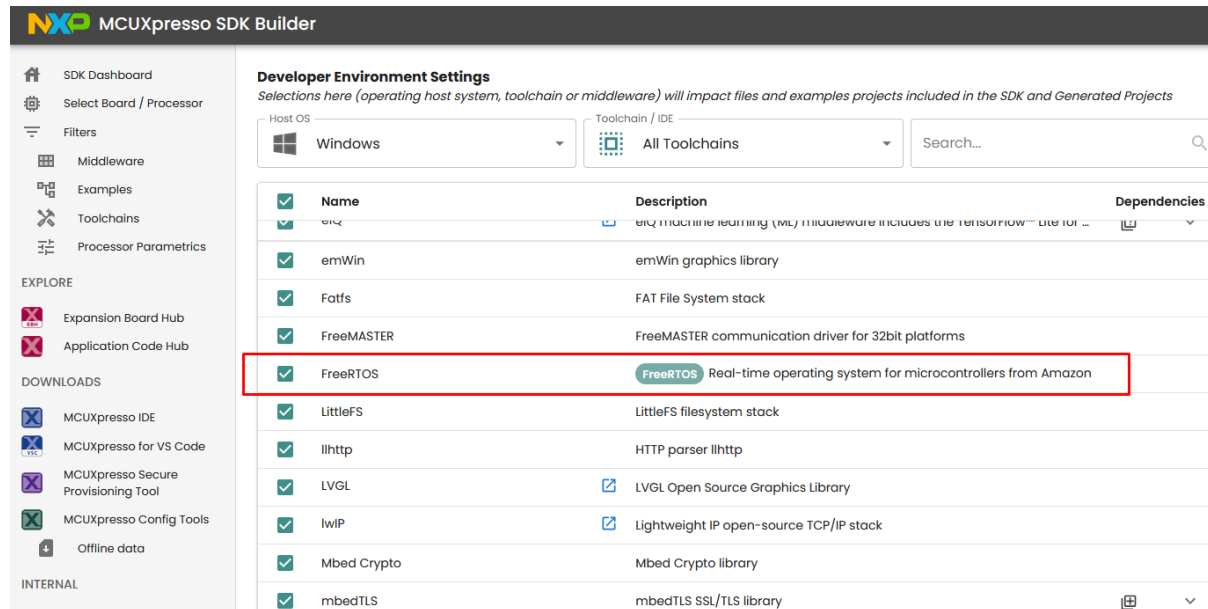
The MCUXpresso SDK framework also contains a set of FreeRTOS examples which show basic FreeRTOS OS features. This makes it easy to start a new FreeRTOS project or begin experimenting with FreeRTOS OS. Selected drivers and middleware are RTOS ready with related FreeRTOS adaptation layer.

**FreeRTOS example applications** The FreeRTOS examples are written to demonstrate basic FreeRTOS features and the interaction between peripheral drivers and the RTOS.

**List of examples** The list of freertos\_examples, their description and availability for individual supported MCUXpresso SDK development boards can be obtained here: [https://mcuxpresso.nxp.com/mcuxsdk/latest/html/examples/freertos\\_examples/index.html](https://mcuxpresso.nxp.com/mcuxsdk/latest/html/examples/freertos_examples/index.html)

**Location of examples** The FreeRTOS examples are located in [mcuxsdk-examples](#) repository, see the `freertos_examples` folder.

Once using MCUXpresso SDK zip packages created via the [MCUXpresso SDK Builder](#) the FreeRTOS kernel library and associated `freertos_examples` are added into final zip package once FreeRTOS components is selected on the Developer Environment Settings page:



The FreeRTOS examples in MCUXpresso SDK zip packages are located in `<MCUXpressoSDK_install_dir>/boards/<board_name>/freertos_examples/` subfolders.

**Building a FreeRTOS example application** For information how to use the cmake and Kconfig based build and configuration system and how to build `freertos_examples` visit: [MCUXpresso SDK documentation for Build And Configuration MCUXpresso SDK Getting Start Guide](#)

Tip: To list all FreeRTOS example projects and targets that can be built via the west build command, use this `west list_project` command in `mcuxsdk` workspace:

```
west list_project -p examples/freertos_examples
```

**FreeRTOS aware debugger plugin** NXP provides FreeRTOS task aware debugger for GDB. The plugin is compatible with Eclipse-based (MCUXpressoIDE) and is available after the installation.

| Task List (FreeRTOS) |           |             |            |          |              |                          |               |
|----------------------|-----------|-------------|------------|----------|--------------|--------------------------|---------------|
| TCB#                 | Task Name | Task Handle | Task State | Priority | Stack Usage  | Event Object             | Runtime       |
| 1                    | task_one  | 0x1fffecc8  | Blocked    | 1 (1)    | 0 B / 880 B  | MyCountingSemaphore (Rx) | 0x0 (0.0%)    |
| 2                    | task_two  | 0x1ffff130  | Blocked    | 2 (2)    | 0 B / 888 B  | MyCountingSemaphore (Rx) | 0x1 (0.1%)    |
| 3                    | IDLE      | 0x1ffff330  | Running    | 0 (0)    | 0 B / 296 B  |                          | 0x3e5 (99.6%) |
| 4                    | Tmr Svc   | 0x1ffff6b8  | Blocked    | 17 (17)  | 28 B / 672 B | TmrQ (Rx)                | 0x3 (0.3%)    |

## FreeRTOS kernel for MCUXpresso SDK ChangeLog

**Changelog FreeRTOS kernel for MCUXpresso 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](#).

**[Unreleased]****Added**

- Kconfig added CONFIG\_FREERTOS\_USE\_CUSTOM\_CONFIG\_FRAGMENT config to optionally include custom FreeRTOSConfig fragment include file FreeRTOSConfig\_frag.h. File must be provided by application.
- Added missing Kconfig option for configUSE\_PICOLIBC\_TLS.
- Add correct header files to build when configUSE\_NEWLIB\_REENTRANT and configUSE\_PICOLIBC\_TLS is selected in config.

**[11.1.0\_rev0]**

- update amazon freertos version

**[11.0.1\_rev0]**

- update amazon freertos version

**[10.5.1\_rev0]**

- update amazon freertos version

**[10.4.3\_rev1]**

- Apply CM33 security fix from 10.4.3-LTS-Patch-2. See rtos\freertos\freertos\_kernel\History.txt
- Apply CM33 security fix from 10.4.3-LTS-Patch-1. See rtos\freertos\freertos\_kernel\History.txt

**[10.4.3\_rev0]**

- update amazon freertos version.

**[10.4.3\_rev0]**

- update amazon freertos version.

**[9.0.0\_rev3]**

- New features:
  - Tickless idle mode support for Cortex-A7. Add fsl\_tickless\_epit.c and fsl\_tickless\_generic.h in portable/IAR/ARM\_CA9 folder.
  - Enabled float context saving in IAR for Cortex-A7. Added configUSE\_TASK\_FPU\_SUPPORT macros. Modified port.c and portmacro.h in portable/IAR/ARM\_CA9 folder.
- Other changes:
  - Transformed ARM\_CM core specific tickless low power support into generic form under freertos/Source/portable/low\_power\_tickless/.



### [9.0.0\_rev2]

- New features:
  - Enabled MCUXpresso thread aware debugging. Add `freertos_tasks_c_additions.h` and `configINCLUDE_FREERTOS_TASK_C_ADDITIONS_H` and `configFREERTOS_MEMORY_SCHEME` macros.

### [9.0.0\_rev1]

- New features:
  - Enabled `-flto` optimization in GCC by adding `attribute((used))` for `vTaskSwitchContext`.
  - Enabled KDS Task Aware Debugger. Apply FreeRTOS patch to enable `configRECORD_STACK_HIGH_ADDRESS` macro. Modified files are `task.c` and `FreeRTOS.h`.

### [9.0.0\_rev0]

- New features:
  - Example `freertos_sem_static`.
  - Static allocation support RTOS driver wrappers.
- Other changes:
  - Tickless idle rework. Support for different timers is in separated files (`fsl_tickless_systick.c`, `fsl_tickless_lptmr.c`).
  - Removed configuration option `configSYSTICK_USE_LOW_POWER_TIMER`. Low power timer is now selected by linking of appropriate file `fsl_tickless_lptmr.c`.
  - Removed `configOVERRIDE_DEFAULT_TICK_CONFIGURATION` in RVDS port. Use of `attribute((weak))` is the preferred solution. Not same as `_weak`!

### [8.2.3]

- New features:
  - Tickless idle mode support.
  - Added template application for Kinetis Expert (KEx) tool (`template_application`).
- Other changes:
  - Folder structure reduction. Keep only Kinetis related parts.

## FreeRTOS kernel Readme

**MCUXpresso SDK: FreeRTOS kernel** This repository is a fork of FreeRTOS kernel (<https://github.com/FreeRTOS/FreeRTOS-Kernel>)(11.1.0). Modifications have been made to adapt to NXP MCUXpresso SDK. `CMakeLists.txt` and `Kconfig` added to enable FreeRTOS kernel repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository `mcuxsdk-manifests`(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

For more information about the FreeRTOS kernel repo adoption see [README\\_mcuxsdk.md: FreeRTOS kernel for MCUXpresso SDK Readme](#) document.



**Getting started** This repository contains FreeRTOS kernel source/header files and kernel ports only. This repository is referenced as a submodule in [FreeRTOS/FreeRTOS](#) repository, which contains pre-configured demo application projects under [FreeRTOS/Demo](#) directory.

The easiest way to use FreeRTOS is to start with one of the pre-configured demo application projects. That way you will have the correct FreeRTOS source files included, and the correct include paths configured. Once a demo application is building and executing you can remove the demo application files, and start to add in your own application source files. See the [FreeRTOS Kernel Quick Start Guide](#) for detailed instructions and other useful links.

Additionally, for FreeRTOS kernel feature information refer to the [Developer Documentation](#), and [API Reference](#).

Also for contributing and creating a Pull Request please refer to *the instructions here*.

**Getting help** If you have any questions or need assistance troubleshooting your FreeRTOS project, we have an active community that can help on the [FreeRTOS Community Support Forum](#).

## To consume FreeRTOS-Kernel

**Consume with CMake** If using CMake, it is recommended to use this repository using FetchContent. Add the following into your project's main or a subdirectory's CMakeLists.txt:

- Define the source and version/tag you want to use:

```
FetchContent_Declare(freertos_kernel
 GIT_REPOSITORY https://github.com/FreeRTOS/FreeRTOS-Kernel.git
 GIT_TAG main #Note: Best practice to use specific git-hash or tagged version
)
```

In case you prefer to add it as a git submodule, do:

```
git submodule add https://github.com/FreeRTOS/FreeRTOS-Kernel.git <path of the submodule>
git submodule update --init
```

- Add a freertos\_config library (typically an INTERFACE library) The following assumes the directory structure:

– include/FreeRTOSConfig.h

```
add_library(freertos_config INTERFACE)

target_include_directories(freertos_config SYSTEM
INTERFACE
 include
)

target_compile_definitions(freertos_config
INTERFACE
 projCOVERAGE_TEST=0
)
```

In case you installed FreeRTOS-Kernel as a submodule, you will have to add it as a subdirectory:

```
add_subdirectory(${FREERTOS_PATH})
```

- Configure the FreeRTOS-Kernel and make it available
  - this particular example supports a native and cross-compiled build option.

```
set(FREERTOS_HEAP "4" CACHE STRING "" FORCE)
Select the native compile PORT
set(FREERTOS_PORT "GCC_POSIX" CACHE STRING "" FORCE)
Select the cross-compile PORT
if (CMAKE_CROSSCOMPILING)
 set(FREERTOS_PORT "GCC_ARM_CA9" CACHE STRING "" FORCE)
endif()

FetchContent_MakeAvailable(freertos_kernel)
```

- In case of cross compilation, you should also add the following to `freertos_config`:

```
target_compile_definitions(freertos_config INTERFACE ${definitions})
target_compile_options(freertos_config INTERFACE ${options})
```

### Consuming stand-alone - Cloning this repository

To clone using HTTPS:

```
git clone https://github.com/FreeRTOS/FreeRTOS-Kernel.git
```

Using SSH:

```
git clone git@github.com:FreeRTOS/FreeRTOS-Kernel.git
```

### Repository structure

- The root of this repository contains the three files that are common to every port - `list.c`, `queue.c` and `tasks.c`. The kernel is contained within these three files. `croutine.c` implements the optional co-routine functionality - which is normally only used on very memory limited systems.
- The `./portable` directory contains the files that are specific to a particular microcontroller and/or compiler. See the readme file in the `./portable` directory for more information.
- The `./include` directory contains the real time kernel header files.
- The `./template_configuration` directory contains a sample `FreeRTOSConfig.h` to help jumpstart a new project. See the *FreeRTOSConfig.h* file for instructions.

**Code Formatting** FreeRTOS files are formatted using the “`uncrustify`” tool. The configuration file used by `uncrustify` can be found in the [FreeRTOS/CI-CD-GitHub-Actions's uncrustify.cfg](#) file.

**Line Endings** File checked into the FreeRTOS-Kernel repository use unix-style LF line endings for the best compatibility with git.

For optimal compatibility with Microsoft Windows tools, it is best to enable the git `autocrlf` feature. You can enable this setting for the current repository using the following command:

```
git config core.autocrlf true
```

**Git History Optimizations** Some commits in this repository perform large refactors which touch many lines and lead to unwanted behavior when using the `git blame` command. You can configure git to ignore the list of large refactor commits in this repository with the following command:

```
git config blame.ignoreRevsFile .git-blame-ignore-revs
```

**Spelling and Formatting** We recommend using [Visual Studio Code](#), commonly referred to as VSCode, when working on the FreeRTOS-Kernel. The FreeRTOS-Kernel also uses [cSpell](#) as part of its spelling check. The config file for which can be found at [cspell.config.yaml](#). There is additionally a [cSpell plugin for VSCode](#) that can be used as well. `.cSpellWords.txt` contains words that are not traditionally found in an English dictionary. It is used by the spellchecker to verify the various jargon, variable names, and other odd words used in the FreeRTOS code base are correct. If your pull request fails to pass the spelling and you believe this is a mistake, then add the word to `.cSpellWords.txt`. When adding a word please then sort the list, which can be done by running the bash command: `sort -u .cSpellWords.txt -o .cSpellWords.txt`. Note that only the FreeRTOS-Kernel Source Files, *include*, *portable/MemMang*, and *portable/Common* files are checked for proper spelling, and formatting at this time.

### 2.1.2 FreeRTOS drivers

This is set of NXP provided FreeRTOS reentrant bus drivers.

### 2.1.3 backoffalgorithm

Algorithm for calculating exponential backoff with jitter for network retry attempts.

#### Readme

**MCUXpresso SDK: backoffAlgorithm Library** This repository is a fork of backoffAlgorithm library (<https://github.com/FreeRTOS/backoffalgorithm>)(1.3.0). Modifications have been made to adapt to NXP MCUXpresso SDK. `CMakeLists.txt` and `Kconfig` added to enable backoffAlgorithm repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository `mcuxsdk-manifests`(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

**backoffAlgorithm Library** This repository contains the backoffAlgorithm library, a utility library to calculate backoff period using an exponential backoff with jitter algorithm for retrying network operations (like failed network connection with server). This library uses the “Full Jitter” strategy for the exponential backoff with jitter algorithm. More information about the algorithm can be seen in the [Exponential Backoff and Jitter](#) AWS blog.

The backoffAlgorithm library is distributed under the *MIT Open Source License*.

Exponential backoff with jitter is typically used when retrying a failed network connection or operation request with the server. An exponential backoff with jitter helps to mitigate failed network operations with servers, that are caused due to network congestion or high request load on the server, by spreading out retry requests across multiple devices attempting network operations. Besides, in an environment with poor connectivity, a client can get disconnected at any time. A backoff strategy helps the client to conserve battery by not repeatedly attempting reconnections when they are unlikely to succeed.

See memory requirements for this library [here](#).

**backoffAlgorithm v1.3.0 source code is part of the FreeRTOS 202210.00 LTS release.**

**backoffAlgorithm v1.0.0 source code is part of the FreeRTOS 202012.00 LTS release.**

**Reference example** The example below shows how to use the backoffAlgorithm library on a POSIX platform to retry a DNS resolution query for amazon.com.

```
#include "backoff_algorithm.h"
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <unistd.h>
#include <time.h>

/* The maximum number of retries for the example code. */
#define RETRY_MAX_ATTEMPTS (5U)

/* The maximum back-off delay (in milliseconds) for between retries in the example. */
#define RETRY_MAX_BACKOFF_DELAY_MS (5000U)

/* The base back-off delay (in milliseconds) for retry configuration in the example. */
#define RETRY_BACKOFF_BASE_MS (500U)

int main()
{
 /* Variables used in this example. */
 BackoffAlgorithmStatus_t retryStatus = BackoffAlgorithmSuccess;
 BackoffAlgorithmContext_t retryParams;
 char serverAddress[] = "amazon.com";
 uint16_t nextRetryBackoff = 0;

 int32_t dnsStatus = -1;
 struct addrinfo hints;
 struct addrinfo ** pListHead = NULL;
 struct timespec tp;

 /* Add hints to retrieve only TCP sockets in getaddrinfo. */
 (void) memset(&hints, 0, sizeof(hints));

 /* Address family of either IPv4 or IPv6. */
 hints.ai_family = AF_UNSPEC;
 /* TCP Socket. */
 hints.ai_socktype = (int32_t) SOCK_STREAM;
 hints.ai_protocol = IPPROTO_TCP;

 /* Initialize reconnect attempts and interval. */
 BackoffAlgorithm_InitializeParams(&retryParams,
 RETRY_BACKOFF_BASE_MS,
 RETRY_MAX_BACKOFF_DELAY_MS,
 RETRY_MAX_ATTEMPTS);

 /* Seed the pseudo random number generator used in this example (with call to
 * rand() function provided by ISO C standard library) for use in backoff period
 * calculation when retrying failed DNS resolution. */

 /* Get current time to seed pseudo random number generator. */
 (void) clock_gettime(CLOCK_REALTIME, &tp);
 /* Seed pseudo random number generator with seconds. */
 srand(tp.tv_sec);

 do
 {
 /* Perform a DNS lookup on the given host name. */
 dnsStatus = getaddrinfo(serverAddress, NULL, &hints, pListHead);
 }
```

(continues on next page)

(continued from previous page)

```

/* Retry if DNS resolution query failed. */
if(dnsStatus != 0)
{
 /* Generate a random number and get back-off value (in milliseconds) for the next retry.
 * Note: It is recommended to use a random number generator that is seeded with
 * device-specific entropy source so that backoff calculation across devices is different
 * and possibility of network collision between devices attempting retries can be avoided.
 *
 * For the simplicity of this code example, the pseudo random number generator, rand()
 * function is used. */
 retryStatus = BackoffAlgorithm_GetNextBackoff(&retryParams, rand(), &nextRetryBackoff);

 /* Wait for the calculated backoff period before the next retry attempt of querying DNS.
 * As usleep() takes nanoseconds as the parameter, we multiply the backoff period by 1000. */
 (void) usleep(nextRetryBackoff * 1000U);
}
} while((dnsStatus != 0) && (retryStatus != BackoffAlgorithmRetriesExhausted));

return dnsStatus;
}

```

**Building the library** A compiler that supports **C90 or later** such as *gcc* is required to build the library.

Additionally, the library uses a header file introduced in ISO C99, *stdint.h*. For compilers that do not provide this header file, the *source/include* directory contains *stdint.readme*, which can be renamed to *stdint.h* to build the backoffAlgorithm library.

For instance, if the example above is copied to a file named *example.c*, *gcc* can be used like so:

```
gcc -I source/include example.c source/backoff_algorithm.c -o example
./example
```

*gcc* can also produce an output file to be linked:

```
gcc -I source/include -c source/backoff_algorithm.c
```

## Building unit tests

**Checkout Unity Submodule** By default, the submodules in this repository are configured with `update=none` in *.gitmodules*, to avoid increasing clone time and disk space usage of other repositories (like [amazon-freertos](#) that submodules this repository).

To build unit tests, the submodule dependency of Unity is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive test/unit-test/Unity
```

## Platform Prerequisites

- For running unit tests
  - C89 or later compiler like *gcc*
  - CMake 3.13.0 or later
- For running the coverage target, *gcov* is additionally required.

### Steps to build Unit Tests

1. Go to the root directory of this repository. (Make sure that the **Unity** submodule is cloned as described [above](#).)
2. Create build directory: `mkdir build && cd build`
3. Run *cmake* while inside build directory: `cmake -S ../test`
4. Run this command to build the library and unit tests: `make all`
5. The generated test executables will be present in `build/bin/tests` folder.
6. Run `ctest` to execute all tests and view the test run summary.

**Contributing** See *CONTRIBUTING.md* for information on contributing.

### 2.1.4 corehttp

C language HTTP client library designed for embedded platforms.

#### MCUXpresso SDK: coreHTTP Client Library

This repository is a fork of coreHTTP Client library (<https://github.com/FreeRTOS/corehttp>)(3.0.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreHTTP Client repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

#### coreHTTP Client Library

This repository contains a C language HTTP client library designed for embedded platforms. It has no dependencies on any additional libraries other than the standard C library, [lhttp](#), and a customer-implemented transport interface. This library is distributed under the *MIT Open Source License*.

This library has gone through code quality checks including verification that no function has a [GNU Complexity](#) score over 8. This library has also undergone both static code analysis from [Coverity static analysis](#), and validation of memory safety and data structure invariance through the [CBMC automated reasoning tool](#).

See memory requirements for this library [here](#).

**coreHTTP v3.0.0 source code is part of the FreeRTOS 202210.00 LTS release.**

**coreHTTP v2.0.0 source code is part of the FreeRTOS 202012.00 LTS release.**

**coreHTTP Config File** The HTTP client library exposes configuration macros that are required for building the library. A list of all the configurations and their default values are defined in *core\_http\_config\_defaults.h*. To provide custom values for the configuration macros, a custom config file named *core\_http\_config.h* can be provided by the user application to the library.

By default, a *core\_http\_config.h* custom config is required to build the library. To disable this requirement and build the library with default configuration values, provide `HTTP_DO_NOT_USE_CUSTOM_CONFIG` as a compile time preprocessor macro.

**The HTTP client library can be built by either:**



- Defining a `core_http_config.h` file in the application, and adding it to the include directories for the library build. **OR**
- Defining the `HTTP_DO_NOT_USE_CUSTOM_CONFIG` preprocessor macro for the library build.

**Building the Library** The `httpFilePaths.cmake` file contains the information of all source files and header include paths required to build the HTTP client library.

As mentioned in the *previous section*, either a custom config file (i.e. `core_http_config.h`) OR `HTTP_DO_NOT_USE_CUSTOM_CONFIG` macro needs to be provided to build the HTTP client library.

For a CMake example of building the HTTP library with the `httpFilePaths.cmake` file, refer to the `coverity_analysis` library target in `test/CMakeLists.txt` file.

## Building Unit Tests

### Platform Prerequisites

- For running unit tests, the following are required:
  - **C90 compiler** like `gcc`
  - **CMake 3.13.0 or later**
  - **Ruby 2.0.0 or later** is required for this repository's [CMock test framework](#).
- For running the coverage target, the following are required:
  - `gcov`
  - `lcov`

### Steps to build Unit Tests

1. Go to the root directory of this repository.
2. Run the `cmake` command: `cmake -S test -B build -DBUILD_CLONE_SUBMODULES=ON`
3. Run this command to build the library and unit tests: `make -C build all`
4. The generated test executables will be present in `build/bin/tests` folder.
5. Run `cd build && ctest` to execute all tests and view the test run summary.

**CBMC** To learn more about CBMC and proofs specifically, review the training material [here](#).

The `test/cbmc/proofs` directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions [here](#).

**Reference examples** The AWS IoT Device SDK for Embedded C repository contains demos of using the HTTP client library [here](#) on a POSIX platform. These can be used as reference examples for the library API.

## Documentation



**Existing Documentation** For pre-generated documentation, please see the documentation linked in the locations below:

| Location                                                       |
|----------------------------------------------------------------|
| <a href="#">AWS IoT Device SDK for Embedded C FreeRTOS.org</a> |

Note that the latest included version of coreHTTP may differ across repositories.

**Generating Documentation** The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

**Contributing** See *CONTRIBUTING.md* for information on contributing.

## 2.1.5 corejson

JSON parser.

### Readme

**MCUXpresso SDK: coreJSON Library** This repository is a fork of coreJSON library (<https://github.com/FreeRTOS/corejson>)(3.2.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreJSON repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

**coreJSON Library** This repository contains the coreJSON library, a parser that strictly enforces the ECMA-404 JSON standard and is suitable for low memory footprint embedded devices. The coreJSON library is distributed under the *MIT Open Source License*.

This library has gone through code quality checks including verification that no function has a [GNU Complexity](#) score over 8, and checks against deviations from mandatory rules in the [MISRA coding standard](#). Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from [Coverity static analysis](#), and validation of memory safety through the [CBMC automated reasoning tool](#).

See memory requirements for this library [here](#).

**coreJSON v3.2.0 source code is part of the FreeRTOS 202210.00 LTS release.**

**coreJSON v3.0.0 source code is part of the FreeRTOS 202012.00 LTS release.**

### Reference example

```

#include <stdio.h>
#include "core_json.h"

int main()
{
 // Variables used in this example.
 JSONStatus_t result;
 char buffer[] = "{\"foo\":\"abc\", \"bar\":{\"foo\":\"xyz\"}}";
 size_t bufferLength = sizeof(buffer) - 1;
 char queryKey[] = "bar.foo";
 size_t queryKeyLength = sizeof(queryKey) - 1;
 char * value;
 size_t valueLength;

 // Calling JSON_Validate() is not necessary if the document is guaranteed to be valid.
 result = JSON_Validate(buffer, bufferLength);

 if(result == JSONSuccess)
 {
 result = JSON_Search(buffer, bufferLength, queryKey, queryKeyLength,
 &value, &valueLength);
 }

 if(result == JSONSuccess)
 {
 // The pointer "value" will point to a location in the "buffer".
 char save = value[valueLength];
 // After saving the character, set it to a null byte for printing.
 value[valueLength] = '\0';
 // "Found: bar.foo -> xyz" will be printed.
 printf("Found: %s -> %s\n", queryKey, value);
 // Restore the original character.
 value[valueLength] = save;
 }

 return 0;
}

```

A search may descend through nested objects when the queryKey contains matching key strings joined by a separator, .. In the example above, bar has the value { "foo": "xyz" }. Therefore, a search for query key bar.foo would output xyz.

**Building coreJSON** A compiler that supports **C90 or later** such as *gcc* is required to build the library.

Additionally, the library uses 2 header files introduced in ISO C99, *stdbool.h* and *stdint.h*. For compilers that do not provide this header file, the *source/include* directory contains *stdbool.readme* and *stdint.readme*, which can be renamed to *stdbool.h* and *stdint.h* respectively.

For instance, if the example above is copied to a file named *example.c*, *gcc* can be used like so:

```
gcc -I source/include example.c source/core_json.c -o example
./example
```

*gcc* can also produce an output file to be linked:

```
gcc -I source/include -c source/core_json.c
```

## Documentation

**Existing documentation** For pre-generated documentation, please see the documentation linked in the locations below:

| Location                                                       |
|----------------------------------------------------------------|
| <a href="#">AWS IoT Device SDK for Embedded C FreeRTOS.org</a> |

Note that the latest included version of the coreJSON library may differ across repositories.

**Generating documentation** The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

## Building unit tests

**Checkout Unity Submodule** By default, the submodules in this repository are configured with `update=none` in `.gitmodules`, to avoid increasing clone time and disk space usage of other repositories (like [amazon-freertos](#) that submodules this repository).

To build unit tests, the submodule dependency of Unity is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive test/unit-test/Unity
```

## Platform Prerequisites

- For running unit tests
  - C90 compiler like gcc
  - CMake 3.13.0 or later
  - Ruby 2.0.0 or later is additionally required for the Unity test framework (that we use).
- For running the coverage target, gcov is additionally required.

## Steps to build Unit Tests

1. Go to the root directory of this repository. (Make sure that the **Unity** submodule is cloned as described [above](#).)
2. Create build directory: `mkdir build && cd build`
3. Run `cmake` while inside build directory: `cmake -S ../test`
4. Run this command to build the library and unit tests: `make all`
5. The generated test executables will be present in `build/bin/tests` folder.
6. Run `ctest` to execute all tests and view the test run summary.

**CBMC** To learn more about CBMC and proofs specifically, review the training material [here](#).

The `test/cbmc/proofs` directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions [here](#).

**Contributing** See *CONTRIBUTING.md* for information on contributing.

## 2.1.6 coremqtt

MQTT publish/subscribe messaging library.

### MCUXpresso SDK: coreMQTT Library

This repository is a fork of coreMQTT library (<https://github.com/FreeRTOS/coremqtt>)(2.1.1). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreMQTT repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

### coreMQTT Client Library

This repository contains the coreMQTT library that has been optimized for a low memory footprint. The coreMQTT library is compliant with the [MQTT 3.1.1](#) standard. It has no dependencies on any additional libraries other than the standard C library, a customer-implemented network transport interface, and *optionally* a user-implemented platform time function. This library is distributed under the *MIT Open Source License*.

This library has gone through code quality checks including verification that no function has a [GNU Complexity](#) score over 8, and checks against deviations from mandatory rules in the [MISRA coding standard](#). Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from [Coverity static analysis](#), and validation of memory safety through the [CBMC automated reasoning tool](#).

See memory requirements for this library [here](#).

**coreMQTT v2.1.1 source code is part of the FreeRTOS 20210.01 LTS release.**

**MQTT Config File** The MQTT client library exposes build configuration macros that are required for building the library. A list of all the configurations and their default values are defined in *core\_mqtt\_config\_defaults.h*. To provide custom values for the configuration macros, a custom config file named *core\_mqtt\_config.h* can be provided by the application to the library.

By default, a *core\_mqtt\_config.h* custom config is required to build the library. To disable this requirement and build the library with default configuration values, provide `MQTT_DO_NOT_USE_CUSTOM_CONFIG` as a compile time preprocessor macro.

**Thus, the MQTT library can be built by either:**

- Defining a *core\_mqtt\_config.h* file in the application, and adding it to the include directories list of the library
- OR**
- Defining the `MQTT_DO_NOT_USE_CUSTOM_CONFIG` preprocessor macro for the library build.

**Sending metrics to AWS IoT** When establishing a connection with AWS IoT, users can optionally report the Operating System, Hardware Platform and MQTT client version information of their device to AWS. This information can help AWS IoT provide faster issue resolution and technical support. If users want to report this information, they can send a specially formatted string (see below) in the username field of the MQTT CONNECT packet.

Format

The format of the username string with metrics is:

```
<Actual_Username>?SDK=<OS_Name>&Version=<OS_Version>&Platform=<Hardware_Platform>&MQTTLib=<MQTT_Library_name>@<MQTT_Library_version>
```

Where

- <Actual\_Username> is the actual username used for authentication, if username and password are used for authentication. When username and password based authentication is not used, this is an empty value.
- <OS\_Name> is the Operating System the application is running on (e.g. FreeRTOS)
- <OS\_Version> is the version number of the Operating System (e.g. V10.4.3)
- <Hardware\_Platform> is the Hardware Platform the application is running on (e.g. WinSim)
- <MQTT\_Library\_name> is the MQTT Client library being used (e.g. coreMQTT)
- <MQTT\_Library\_version> is the version of the MQTT Client library being used (e.g. 1.0.2)

Example

- Actual\_Username = "iotuser", OS\_Name = FreeRTOS, OS\_Version = V10.4.3, Hardware\_Platform\_Name = WinSim, MQTT\_Library\_Name = coremqtt, MQTT\_Library\_version = 2.1.1. If username is not used, then "iotuser" can be removed.

```
/* Username string:
 * iotuser?SDK=FreeRTOS&Version=v10.4.3&Platform=WinSim&MQTTLib=coremqtt@2.1.1
 */

#define OS_NAME "FreeRTOS"
#define OS_VERSION "V10.4.3"
#define HARDWARE_PLATFORM_NAME "WinSim"
#define MQTT_LIB "coremqtt@2.1.1"

#define USERNAME_STRING "iotuser?SDK=" OS_NAME "&Version=" OS_VERSION "&Platform=" HARDWARE_PLATFORM_NAME "&MQTTLib=" MQTT_LIB
#define USERNAME_STRING_LENGTH ((uint16_t) (sizeof(USERNAME_STRING) - 1))

MQTTConnectInfo_t connectInfo;
connectInfo.userName = USERNAME_STRING;
connectInfo.userNameLength = USERNAME_STRING_LENGTH;
mqttStatus = MQTT_Connect(pMqttContext, &connectInfo, NULL, CONNACK_RECV_TIMEOUT_MS,
↳ pSessionPresent);
```

**Upgrading to v2.0.0 and above** With coreMQTT versions >=v2.0.0, there are breaking changes. Please refer to the *coreMQTT version >=v2.0.0 Migration Guide*.

**Building the Library** The *mqttFilePaths.cmake* file contains the information of all source files and the header include path required to build the MQTT library.

Additionally, the MQTT library requires two header files that are not part of the ISO C90 standard library, *stdbool.h* and *stdint.h*. For compilers that do not provide these header files, the

*source/include* directory contains the files *stdbool.readme* and *stdint.readme*, which can be renamed to *stdbool.h* and *stdint.h*, respectively, to provide the type definitions required by MQTT.

As mentioned in the previous section, either a custom config file (i.e. *core\_mqtt\_config.h*) OR `MQTT_DO_NOT_USE_CUSTOM_CONFIG` macro needs to be provided to build the MQTT library.

For a CMake example of building the MQTT library with the *mqttFilePaths.cmake* file, refer to the *coverity\_analysis* library target in *test/CMakeLists.txt* file.

## Building Unit Tests

**Checkout CMock Submodule** By default, the submodules in this repository are configured with `update=none` in *.gitmodules* to avoid increasing clone time and disk space usage of other repositories (like [amazon-freertos](#) that submodules this repository).

To build unit tests, the submodule dependency of CMock is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive test/unit-test/CMock
```

## Platform Prerequisites

- Docker

or the following:

- For running unit tests
  - **C90 compiler** like gcc
  - **CMake 3.13.0 or later**
  - **Ruby 2.0.0 or later** is additionally required for the CMock test framework (that we use).
- For running the coverage target, **gcov** and **lcov** are additionally required.

## Steps to build Unit Tests

1. If using docker, launch the container:
  1. `docker build -t coremqtt .`
  2. `docker run -it -v "$PWD":/workspaces/coreMQTT -w /workspaces/coreMQTT coremqtt`
2. Go to the root directory of this repository. (Make sure that the **CMock** submodule is cloned as described [above](#))
3. Run the *cmake* command: `cmake -S test -B build`
4. Run this command to build the library and unit tests: `make -C build all`
5. The generated test executables will be present in *build/bin/tests* folder.
6. Run `cd build && ctest` to execute all tests and view the test run summary.

**CBMC** To learn more about CBMC and proofs specifically, review the training material [here](#).

The *test/cbmc/proofs* directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions [here](#).

**Reference examples** Please refer to the demos of the MQTT client library in the following locations for reference examples on POSIX and FreeRTOS platforms:

| Platform | Location                                            | Transport Interface Implementation                 |
|----------|-----------------------------------------------------|----------------------------------------------------|
| POSIX    | <a href="#">AWS IoT Device SDK for Embedded C</a>   | POSIX sockets for TCP/IP and OpenSSL for TLS stack |
| FreeRTOS | <a href="#">FreeRTOS/FreeRTOS</a>                   | FreeRTOS+TCP for TCP/IP and mbedTLS for TLS stack  |
| FreeRTOS | <a href="#">FreeRTOS AWS Reference Integrations</a> | Based on Secure Sockets Abstraction                |

## Documentation

**Existing Documentation** For pre-generated documentation, please see the documentation linked in the locations below:

| Location                                          |
|---------------------------------------------------|
| <a href="#">AWS IoT Device SDK for Embedded C</a> |
| <a href="#">FreeRTOS.org</a>                      |

Note that the latest included version of coreMQTT may differ across repositories.

**Generating Documentation** The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

**Contributing** See *CONTRIBUTING.md* for information on contributing.

## 2.1.7 coremqtt-agent

The coreMQTT Agent library is a high level API that adds thread safety to the coreMQTT library.

### Readme

**MCUXpresso SDK: coreMQTT Agent Library** This repository is a fork of coreMQTT Agent library (<https://github.com/FreeRTOS/coremqtt-agent>)(1.2.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreMQTT Agent repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.



**coreMQTT Agent Library** The coreMQTT Agent library is a high level API that adds thread safety to the [coreMQTT](#) library. The library provides thread safe equivalents to the coreMQTT's APIs, greatly simplifying its use in multi-threaded environments. The coreMQTT Agent library manages the MQTT connection by serializing the access to the coreMQTT library and reducing implementation overhead (e.g., removing the need for the application to repeatedly call to MQTT\_ProcessLoop). This allows your multi-threaded applications to share the same MQTT connection, and enables you to design an embedded application without having to worry about coreMQTT thread safety.

This library has gone through code quality checks including verification that no function has a [GNU Complexity](#) score over 8, and checks against deviations from mandatory rules in the [MISRA coding standard](#). Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from [Coverity static analysis](#), and validation of memory safety through the [CBMC automated reasoning tool](#).

See memory requirements for this library [here](#).

**Cloning this repository** This repo uses [Git Submodules](#) to bring in dependent components.

To clone using HTTPS:

```
git clone https://github.com/FreeRTOS/coreMQTT-Agent.git --recurse-submodules
```

Using SSH:

```
git clone git@github.com:FreeRTOS/coreMQTT-Agent.git --recurse-submodules
```

If you have downloaded the repo without using the `--recurse-submodules` argument, you need to run:

```
git submodule update --init --recursive
```

**coreMQTT Agent Library Configurations** The MQTT Agent library uses the same `core_mqtt_config.h` configuration file as coreMQTT, with the addition of configuration constants listed at the top of `core_mqtt_agent.h` and `core_mqtt_agent_command_functions.h`. Documentation for these configurations can be found [here](#).

To provide values for these configuration values, they must be either:

- Defined in `core_mqtt_config.h` used by coreMQTT **OR**
- Passed as compile time preprocessor macros

**Porting the coreMQTT Agent Library** In order to use the MQTT Agent library on a platform, you need to supply thread safe functions for the agent's *messaging interface*.

**Messaging Interface** Each of the following functions must be thread safe.



| Function Pointer          | Description                                                                                                                                                                                                                                                                   |
|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| MQTTAgentMessageSend_t    | A function that sends commands (as MQTTAgentCommand_t * pointers) to be received by MQTTAgent_CommandLoop. This can be implemented by pushing to a thread safe queue.                                                                                                         |
| MQTTAgentMessageRecv_t    | A function used by MQTTAgent_CommandLoop to receive MQTTAgentCommand_t * pointers that were sent by API functions. This can be implemented by receiving from a thread safe queue.                                                                                             |
| MQTTAgentCommandGet_t     | A function that returns a pointer to an allocated MQTTAgentCommand_t structure, which is used to hold information and arguments for a command to be executed in MQTTAgent_CommandLoop(). If using dynamic memory, this can be implemented using malloc().                     |
| MQTTAgentCommandRelease_t | A function called to indicate that a command structure that had been allocated with the MQTTAgentCommandGet_t function pointer will no longer be used by the agent, so it may be freed or marked as not in use. If using dynamic memory, this can be implemented with free(). |

Reference implementations for the interface functions can be found in the [reference examples](#) below.

## Additional Considerations

**Static Memory** If only static allocation is used, then the MQTTAgentCommandGet\_t and MQTTAgentCommandRelease\_t could instead be implemented with a pool of MQTTAgentCommand\_t structures, with a queue or semaphore used to control access and provide thread safety. The below [reference examples](#) use static memory with a command pool.

**Subscription Management** The MQTT Agent does not track subscriptions for MQTT topics. The receipt of any incoming PUBLISH packet will result in the invocation of a single MQTTAgentIncomingPublishCallback\_t callback, which is passed to MQTTAgent\_Init() for initialization. If it is desired for different handlers to be invoked for different incoming topics, then the publish callback will have to manage subscriptions and fan out messages. A platform independent subscription manager example is implemented in the [reference examples](#) below.

**Building the Library** You can build the MQTT Agent source files that are in the *source* directory, and add *source/include* to your compiler's include path. Additionally, the MQTT Agent library requires the coreMQTT library, whose files follow the same *source/* and *source/include* pattern as the agent library; its build instructions can be found [here](#).

If using CMake, the *mqttAgentFilePaths.cmake* file contains the above information of the source files and the header include path from this repository. The same information is found for coreMQTT from *mqttFilePaths.cmake* in the *coreMQTT submodule*.

For a CMake example of building the MQTT Agent library with the *mqttAgentFilePaths.cmake* file, refer to the *coverity\_analysis* library target in *test/CMakeLists.txt* file.

## Building Unit Tests

**Checkout CMock Submodule** To build unit tests, the submodule dependency of CMock is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive test/unit-test/CMock
```

### Unit Test Platform Prerequisites

- For running unit tests
  - **C90 compiler** like gcc
  - **CMake 3.13.0 or later**
  - **Ruby 2.0.0 or later** is additionally required for the CMock test framework (that we use).
- For running the coverage target, **gcov** and **lcov** are additionally required.

### Steps to build Unit Tests

1. Go to the root directory of this repository. (Make sure that the **CMock** submodule is cloned as described [above](#))
2. Run the *cmake* command: `cmake -S test -B build`
3. Run this command to build the library and unit tests: `make -C build all`
4. The generated test executables will be present in `build/bin/tests` folder.
5. Run `cd build && ctest` to execute all tests and view the test run summary.

**CBMC** To learn more about CBMC and proofs specifically, review the training material [here](#).

The `test/cbmc/proofs` directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions [here](#).

**Reference examples** Please refer to the demos of the MQTT Agent library in the following locations for reference examples on FreeRTOS platforms:

| Location                             |
|--------------------------------------|
| <a href="#">coreMQTT Agent Demos</a> |
| <a href="#">FreeRTOS/FreeRTOS</a>    |

**Documentation** The MQTT Agent API documentation can be found [here](#).

**Generating documentation** The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages yourself, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

**Getting help** You can use your Github login to get support from both the FreeRTOS community and directly from the primary FreeRTOS developers on our [active support forum](#). You can find a list of [frequently asked questions](#) [here](#).

**Contributing** See *CONTRIBUTING.md* for information on contributing.

**License** This library is licensed under the MIT License. See the *LICENSE* file.

## 2.1.8 corepkcs11

PKCS #11 key management library.

### Readme

**MCUXpresso SDK: corePKCS11 Library** This repository is a fork of PKCS #11 key management library (<https://github.com/FreeRTOS/corePKCS11/tree/v3.5.0>)(v3.5.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable corepkcs11 repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

**corePKCS11 Library** PKCS #11 is a standardized and widely used API for manipulating common cryptographic objects. It is important because the functions it specifies allow application software to use, create, modify, and delete cryptographic objects, without ever exposing those objects to the application's memory. For example, FreeRTOS AWS reference integrations use a small subset of the PKCS #11 API to, among other things, access the secret (private) key necessary to create a network connection that is authenticated and secured by the [Transport Layer Security \(TLS\)](#) protocol – without the application ever ‘seeing’ the key.

The Cryptoki or PKCS #11 standard defines a platform-independent API to manage and use cryptographic tokens. The name, “PKCS #11”, is used interchangeably to refer to the API itself and the standard which defines it.

This repository contains a software based mock implementation of the PKCS #11 interface (API) that uses the cryptographic functionality provided by Mbed TLS. Using a software mock enables rapid development and flexibility, but it is expected that the mock be replaced by an implementation specific to your chosen secure key storage in production devices.

Only a subset of the PKCS #11 standard is implemented, with a focus on operations involving asymmetric keys, random number generation, and hashing.

The targeted use cases include certificate and key management for TLS authentication and code-sign signature verification, on small embedded devices.

corePKCS11 is implemented on PKCS #11 v2.4.0, the full PKCS #11 standard can be found on the [oasis website](#).

This library has gone through code quality checks including verification that no function has a [GNU Complexity](#) score over 8, and checks against deviations from mandatory rules in the [MISRA coding standard](#). Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from [Coverity static analysis](#) and validation of memory safety through the [CBMC automated reasoning tool](#).

See memory requirements for this library [here](#).

**corePKCS11 v3.5.0 source code is part of the FreeRTOS 202210.00 LTS release.**

**corePKCS11 v3.0.0 source code is part of the FreeRTOS 202012.00 LTS release.**

**Purpose** Generally vendors for secure cryptoprocessors such as Trusted Platform Module (TPM), Hardware Security Module (HSM), Secure Element, or any other type of secure hardware enclave, distribute a PKCS #11 implementation with the hardware. The purpose of the corePKCS11 software only mock library is therefore to provide a non hardware specific PKCS #11 implementation that allows for rapid prototyping and development before switching to a cryptoprocessor specific PKCS #11 implementation in production devices.

Since the PKCS #11 interface is defined as part of the PKCS #11 [specification](#) replacing this library with another implementation should require little porting effort, as the interface will not change. The system tests distributed in this repository can be leveraged to verify the behavior of a different implementation is similar to corePKCS11.

**corePKCS11 Configuration** The corePKCS11 library exposes preprocessor macros which must be defined prior to building the library. A list of all the configurations and their default values are defined in the doxygen documentation for this library.

## Build Prerequisites

**Library Usage** For building the library the following are required:

- **A C99 compiler**
- **mbedcrypto** library from [mbedtls](#) version 2.x or 3.x.
- **pkcs11 API header(s)** available from [OASIS](#) or [OpenSC](#)

Optionally, variables from the pkcsFilePaths.cmake file may be referenced if your project uses cmake.

**Integration and Unit Tests** In order to run the integration and unit test suites the following are dependencies are necessary:

- **C Compiler**
- **CMake 3.13.0 or later**
- **Ruby 2.0.0 or later** required by CMock.
- **Python 3** required for configuring mbedtls.
- **git** required for fetching dependencies.
- **GNU Make** or **Ninja**

The *mbedtls*, *CMock*, and *Unity* libraries are downloaded and built automatically using the cmake FetchContent feature.

**Coverage Measurement and Instrumentation** The following software is required to run the coverage target:

- Linux, MacOS, or another POSIX-like environment.
- A recent version of **GCC** or **Clang** with support for gcov-like coverage instrumentation.
- **gcov** binary corresponding to your chosen compiler
- **lcov** from the [Linux Test Project](#)
- **perl** needed to run the lcov utility.

Coverage builds are validated on recent versions of Ubuntu Linux.

## Running the Integration and Unit Tests

1. Navigate to the root directory of this repository in your shell.
2. Run **cmake** to construct a build tree: `cmake -S test -B build`
  - You may specify your preferred build tool by appending `-G'Unix Makefiles'` or `-GNinja` to the command above.
  - You may append `-DUNIT_TESTS=0` or `-DSYSTEM_TESTS=0` to disable Unit Tests or Integration Tests respectively.
3. Build the test binaries: `cmake --build ./build --target all`
4. Run `ctest --test-dir ./build` or `cmake --build ./build --target test` to run the tests without capturing coverage.
5. Run `cmake --build ./build --target coverage` to run the tests and capture coverage data.

**CBMC** To learn more about CBMC and proofs specifically, review the training material [here](#).

The `test/cbmc/proofs` directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions [here](#).

**Reference examples** The FreeRTOS-Labs repository contains demos using the PKCS #11 library [here](#) using FreeRTOS on the Windows simulator platform. These can be used as reference examples for the library API.

**Porting Guide** Documentation for porting corePKCS11 to a new platform can be found on the AWS [docs](#) web page.

corePKCS11 is not meant to be ported to projects that have a TPM, HSM, or other hardware for offloading crypto-processing. This library is specifically meant to be used for development and prototyping.

**Related Example Implementations** These projects implement the PKCS #11 interface on real hardware and have similar behavior to corePKCS11. It is preferred to use these, over corePKCS11, as they allow for offloading Cryptography to separate hardware.

- ARM's [Platform Security Architecture](#).
- Microchip's [cryptoauthlib](#).
- Infineon's [Optiga Trust X](#).

## Documentation

**Existing Documentation** For pre-generated documentation, please see the documentation linked in the locations below:

| Location                                                       |
|----------------------------------------------------------------|
| <a href="#">AWS IoT Device SDK for Embedded C FreeRTOS.org</a> |

Note that the latest included version of corePKCS11 may differ across repositories.

**Generating Documentation** The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

**Security** See *CONTRIBUTING* for more information.

**License** This library is licensed under the MIT-0 License. See the LICENSE file.

### 2.1.9 freertos-plus-tcp

Open source RTOS FreeRTOS Plus TCP.

#### Readme

**MCUXpresso SDK: FreeRTOS-Plus-TCP Library** This repository is a fork of FreeRTOS-Plus-TCP library (<https://github.com/FreeRTOS/freertos-plus-tcp>)(4.0.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable FreeRTOS-Plus-TCP repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

**Introduction** This branch contains unified IPv4 and IPv6 functionalities. Refer to the Getting started Guide (found [here](#)) for more details.

**FreeRTOS-Plus-TCP Library** FreeRTOS-Plus-TCP is a lightweight TCP/IP stack for FreeRTOS. It provides a familiar Berkeley sockets interface, making it as simple to use and learn as possible. FreeRTOS-Plus-TCP's features and RAM footprint are fully scalable, making FreeRTOS-Plus-TCP equally applicable to smaller lower throughput microcontrollers as well as larger higher throughput microprocessors.

This library has undergone static code analysis and checks for compliance with the [MISRA coding standard](#). Any deviations from the MISRA C:2012 guidelines are documented under [MISRA Deviations](#). The library is validated for memory safety and data structure invariance through the [CBMC automated reasoning tool](#) for the functions that parse data originating from the network. The library is also protocol tested using Maxwell protocol tester for both IPv4 and IPv6.

**Getting started** The easiest way to use the 4.0.0 version of FreeRTOS-Plus-TCP is to refer the Getting started Guide (found [here](#)) Another way is to start with the pre-configured demo application project (found in [this directory](#)). That way you will have the correct FreeRTOS source files included, and the correct include paths configured. Once a demo application is building and executing you can remove the demo application files, and start to add in your own application source files. See the [FreeRTOS Kernel Quick Start Guide](#) for detailed instructions and other useful links.

Additionally, for FreeRTOS-Plus-TCP source code organization refer to the [Documentation](#), and [API Reference](#).



**Getting help** If you have any questions or need assistance troubleshooting your FreeRTOS project, we have an active community that can help on the [FreeRTOS Community Support Forum](#). Please also refer to [FAQ](#) for frequently asked questions.

Also see the [Submitting a bugs/feature request](#) section of CONTRIBUTING.md for more details.

**Note:** All the remaining sections are generic and applies to all the versions from V3.0.0 onwards.

**Upgrading to V3.0.0 and V3.1.0** In version 3.0.0 or 3.1.0, the folder structure of FreeRTOS-Plus-TCP has changed and the files have been broken down into smaller logically separated modules. This change makes the code more modular and conducive to unit-tests. FreeRTOS-Plus-TCP V3.0.0 improves the robustness, security, and modularity of the library. Version 3.0.0 adds comprehensive unit test coverage for all lines and branches of code and has undergone protocol testing, and penetration testing by AWS Security to reduce the exposure to security vulnerabilities. Additionally, the source files have been moved to a `source` directory. This change requires modification of any existing project(s) to include the modified source files and directories. There are examples on how to use the new files and directory structure. For an example based on the Xilinx Zynq-7000, use the code in this [branch](#) and follow these [instructions](#) to build and run the demo.

**FreeRTOS-Plus-TCP V3.1.0 source code(.c .h) is part of the FreeRTOS 202210.00 LTS release.**

**Generating pre V3.0.0 folder structure for backward compatibility:** If you wish to continue using a version earlier than V3.0.0 i.e. continue to use your existing source code organization, a script is provided to generate the folder structure similar to [this](#).

**Note:** After running the script, while the `.c` files will have same names as the pre V3.0.0 source, the files in the `include` directory will have different names and the number of files will differ as well. This should, however, not pose any problems to most projects as projects generally include all files in a given directory.

Running the script to generate pre V3.0.0 folder structure: For running the script, you will need Python version > 3.7. You can download/install it from [here](#).

Once python is downloaded and installed, you can verify the version from your terminal/command window by typing `python --version`.

To run the script, you should switch to the FreeRTOS-Plus-TCP directory that was created using the *Cloning this repository* step above. And then run `python <Path/to/the/script>/GenerateOriginalFiles.py`.

## To consume FreeRTOS+TCP

**Consume with CMake** If using CMake, it is recommended to use this repository using FetchContent. Add the following into your project's main or a subdirectory's CMakeLists.txt:

- Define the source and version/tag you want to use:

```
FetchContent_Declare(freertos_plus_tcp
 GIT_REPOSITORY https://github.com/FreeRTOS/FreeRTOS-Plus-TCP.git
 GIT_TAG master #Note: Best practice to use specific git-hash or tagged version
 GIT_SUBMODULES "" # Don't grab any submodules since not latest
)
```

- Configure the FreeRTOS-Kernel and make it available
  - this particular example supports a native and cross-compiled build option.



```

set(FREERTOS_PLUS_FAT_DEV_SUPPORT OFF CACHE BOOL "" FORCE)
Select the native compile PORT
set(FREERTOS_PLUS_FAT_PORT "POSIX" CACHE STRING "" FORCE)
Select the cross-compile PORT
if (CMAKE_CROSSCOMPILING)
 # Eg. Zynq 2019_3 version of port
 set(FREERTOS_PLUS_FAT_PORT "ZYNQ_2019_3" CACHE STRING "" FORCE)
endif()

FetchContent_MakeAvailable(freertos_plus_tcp)

```

**Consuming stand-alone** This repository uses [Git Submodules](#) to bring in dependent components.

Note: If you download the ZIP file provided by GitHub UI, you will not get the contents of the submodules. (The ZIP file is also not a valid Git repository)

To clone using HTTPS:

```

git clone https://github.com/FreeRTOS/FreeRTOS-Plus-TCP.git ./FreeRTOS-Plus-TCP
cd ./FreeRTOS-Plus-TCP
git submodule update --checkout --init --recursive tools/CMock test/FreeRTOS-Kernel

```

Using SSH:

```

git clone git@github.com:FreeRTOS/FreeRTOS-Plus-TCP.git ./FreeRTOS-Plus-TCP
cd ./FreeRTOS-Plus-TCP
git submodule update --checkout --init --recursive tools/CMock test/FreeRTOS-Kernel

```

**Porting** The porting guide is available on [this page](#).

**Repository structure** This repository contains the FreeRTOS-Plus-TCP repository and a number of supplementary libraries for testing/PR Checks. Below is the breakdown of what each directory contains:

- tools
  - This directory contains the tools and related files (CMock/uncrustify) required to run tests/checks on the TCP source code.
- tests
  - This directory contains all the tests (unit tests and CBMC) and the dependencies ([FreeRTOS-Kernel/Litani-port](#)) the tests require.
- source/portable
  - This directory contains the portable files required to compile the FreeRTOS-Plus-TCP source code for different hardware/compilers.
- source/include
  - The include directory has all the ‘core’ header files of FreeRTOS-Plus-TCP source.
- source
  - This directory contains all the [.c] source files.

**Note** At this time it is recommended to use BufferAllocation\_2.c in which case it is essential to use the heap\_4.c memory allocation scheme. See [memory management](#).

**Kernel sources** The FreeRTOS Kernel Source is in [FreeRTOS/FreeRTOS-Kernel repository](#), and it is consumed by testing/PR checks as a submodule in this repository.

The version of the FreeRTOS Kernel Source in use could be accessed at `./test/FreeRTOS-Kernel` directory.

**CBMC** The `test/cbmc/proofs` directory contains CBMC proofs.

To learn more about CBMC and proofs specifically, review the training material [here](#).

In order to run these proofs you will need to install CBMC and other tools by following the instructions [here](#).