



MCUXpresso SDK Documentation

Release 25.12.00



NXP
Dec 18, 2025



Table of contents

1	TWR-KM35Z75M	3
1.1	Overview	3
1.2	Getting Started with MCUXpresso SDK Package	3
1.2.1	Getting Started with Package	3
1.3	Getting Started with MCUXpresso SDK GitHub	22
1.3.1	Getting Started with MCUXpresso SDK Repository	22
1.4	Release Notes	46
1.4.1	MCUXpresso SDK Release Notes	46
1.5	ChangeLog	49
1.5.1	MCUXpresso SDK Changelog	49
1.6	Driver API Reference Manual	92
1.7	Middleware Documentation	92
1.7.1	FreeMASTER	92
1.7.2	FreeRTOS	92
2	MKM35Z7	93
2.1	ADC16: 16-bit SAR Analog-to-Digital Converter Driver	93
2.2	AFE: Analog Front End Driver	102
2.3	Clock Driver	110
2.4	CMP: Analog Comparator Driver	134
2.5	CRC: Cyclic Redundancy Check Driver	139
2.6	DMA: Direct Memory Access Controller Driver	142
2.7	DMAMUX: Direct Memory Access Multiplexer Driver	152
2.8	EWM: External Watchdog Monitor Driver	153
2.9	FGPIO Driver	156
2.10	C90TFS Flash Driver	158
2.11	ftfx adapter	158
2.12	Ftftx CACHE Driver	158
2.13	ftfx controller	160
2.14	ftfx feature	176
2.15	Ftftx FLASH Driver	177
2.16	Ftftx FLEXNVM Driver	190
2.17	ftfx utilities	201
2.18	GPIO: General-Purpose Input/Output Driver	202
2.19	GPIO Driver	203
2.20	I2C: Inter-Integrated Circuit Driver	205
2.21	I2C DMA Driver	205
2.22	I2C Driver	207
2.23	IRTC: IRTC Driver	221
2.24	Common Driver	232
2.25	Lin_lpuart_driver	245
2.26	LLWU: Low-Leakage Wakeup Unit Driver	253
2.27	LPTMR: Low-Power Timer	256
2.28	LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver	262
2.29	LPUART DMA Driver	262
2.30	LPUART Driver	265

2.31	MCM: Miscellaneous Control Module	283
2.32	MMAU: Memory Mapped Arithmetic Unit	288
2.33	PDB: Programmable Delay Block	318
2.34	PIT: Periodic Interrupt Timer	328
2.35	PMC: Power Management Controller	332
2.36	PORT: Port Control and Interrupts	338
2.37	QTMR: Quad Timer Driver	345
2.38	RCM: Reset Control Module Driver	353
2.39	RNGA: Random Number Generator Accelerator Driver	358
2.40	SIM: System Integration Module Driver	360
2.41	SLCD: Segment LCD Driver	361
2.42	Smart Card	373
2.43	Smart Card UART Driver	381
2.44	SMC: System Mode Controller Driver	384
2.45	SPI: Serial Peripheral Interface Driver	390
2.46	SPI DMA Driver	390
2.47	SPI Driver	393
2.48	SYSMPU: System Memory Protection Unit	406
2.49	UART: Universal Asynchronous Receiver/Transmitter Driver	414
2.50	UART DMA Driver	414
2.51	UART Driver	416
2.52	VREF: Voltage Reference Driver	432
2.53	WDOG: Watchdog Timer Driver	435
2.54	XBAR: Inter-Peripheral Crossbar Switch	442
3	Middleware	445
3.1	Motor Control	445
3.1.1	FreeMASTER	445
4	RTOS	483
4.1	FreeRTOS	483
4.1.1	FreeRTOS kernel	483
4.1.2	FreeRTOS drivers	483
4.1.3	backoffalgorithm	483
4.1.4	corehttp	483
4.1.5	corejson	483
4.1.6	coremqtt	484
4.1.7	corepkcs11	484
4.1.8	freertos-plus-tcp	484

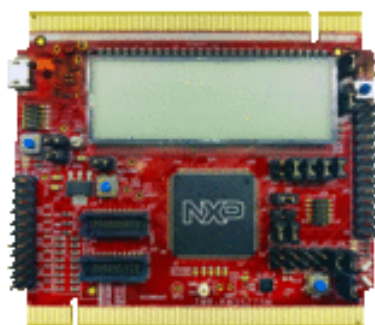
This documentation contains information specific to the twrk35z75m board.

Chapter 1

TWR-KM35Z75M

1.1 Overview

The TWR-KM35Z75M board is designed to work either in standalone mode or as part of the Tower System, a modular development board platform that enables rapid prototyping and tool re-use through reconfigurable hardware. Begin constructing your Tower System evaluation board platform today by visiting NXP.com/Tower for additional Tower System boards and compatible peripherals.



MCU device and part on board is shown below:

- Device: MKM35Z7
- PartNumber: MKM35Z512VLQ7

1.2 Getting Started with MCUXpresso SDK Package

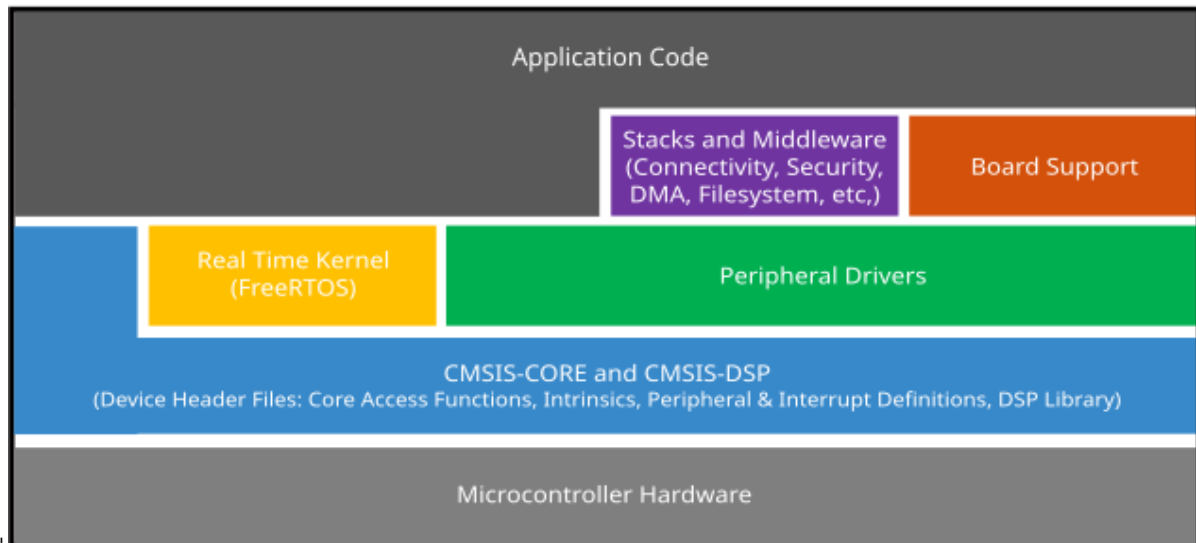
1.2.1 Getting Started with Package

Overview

The NXP MCUXpresso software and tools offer comprehensive development solutions designed to optimize, ease, and help accelerate embedded system development of applications based on general purpose, crossover, and Bluetooth™-enabled MCUs from NXP. The MCUXpresso SDK includes a flexible set of peripheral drivers designed to speed up and simplify development of embedded applications. Along with the peripheral drivers, the MCUXpresso SDK provides an extensive and rich set of example applications covering everything from basic peripheral use case examples to full demo applications. The MCUXpresso SDK contains optional RTOS integrations such as FreeRTOS and Azure RTOS, and various other middleware to support rapid development.

For supported toolchain versions, see *MCUXpresso SDK Release Notes for TWR-KM35Z75M* (document MCUXSDKKM35Z75MRN).

For more details about MCUXpresso SDK, see the MCUXpresso SDK homepage [MCUXpresso-SDK: Software Development Kit for MCUXpresso](#).



MCUXpresso SDK board support package folders

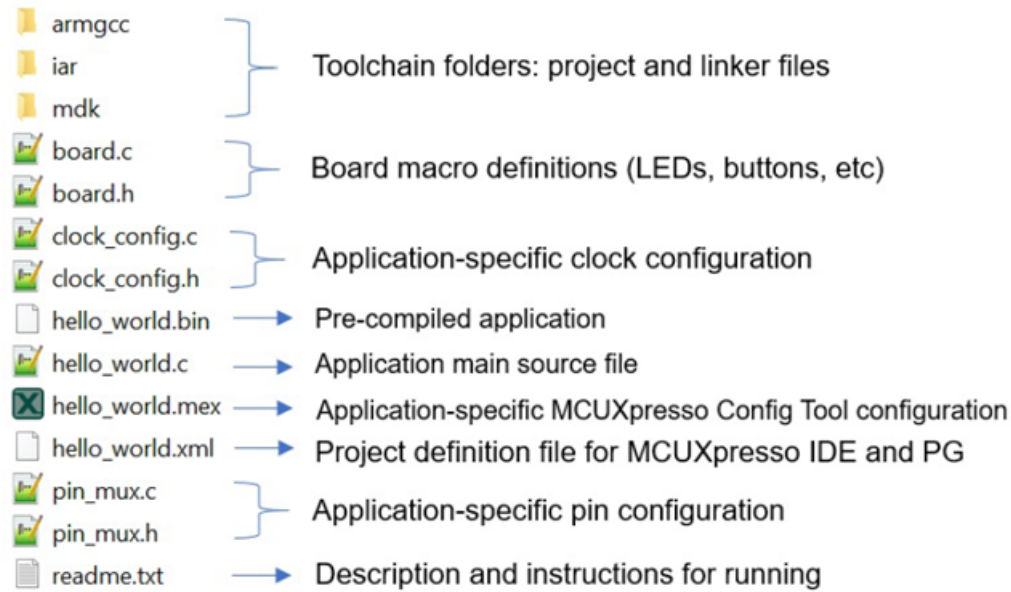
MCUXpresso SDK board support package provides example applications for NXP development and evaluation boards for Arm Cortex-M cores including Freedom, Tower System, and LPCXpresso boards. Board support packages are found inside the top-level boards folder and each supported board has its own folder (an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

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

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

Each `<board_name>` folder in the boards directory contains a comprehensive set of examples that are relevant to that specific piece of hardware. Although we use the `hello_world` example (part of the `demo_apps` folder), the same general rules apply to any type of example in the `<board_name>` folder.

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



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

Parent topic: [MCUXpresso SDK board support package folders](#)

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

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

- `devices/<device_name>`: The device's CMSIS header file, MCUXpresso SDK feature file, and a few other things.
- `devices/<device_name>/cmsis_drivers`: All the CMSIS drivers for your specific MCU.
- `devices/<device_name>/drivers`: All of the peripheral drivers for your specific MCU.
- `devices/<device_name>/<tool_name>`: Toolchain-specific startup code. Vector table definitions are here.
- `devices/<device_name>/utilities`: Items such as the debug console that are used by many of the example applications.
- `devices/<device_name>/project template`: Project template used in CMSIS PACK new project creation.

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

Parent topic: [MCUXpresso SDK board support package folders](#)

Run a demo using MCUXpresso IDE

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

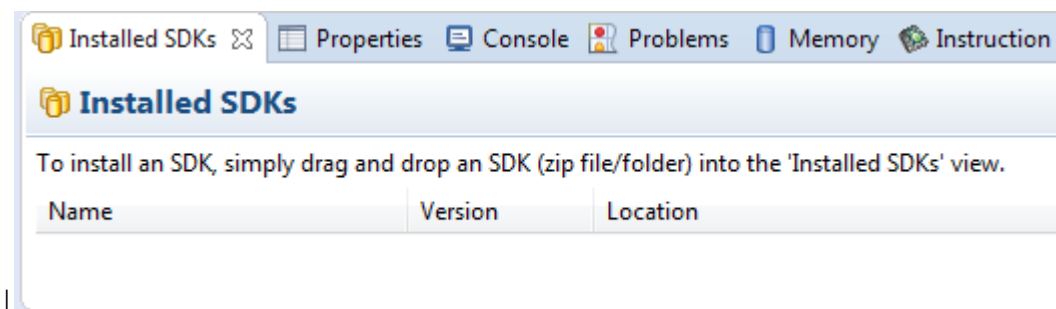
This section describes the steps required to configure MCUXpresso IDE v11.1.1 to build, run, and debug example applications. The `hello_world` demo application targeted for the TWR-KM35Z75M hardware platform is used as an example, though these steps can be applied to any example application in the MCUXpresso SDK.

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

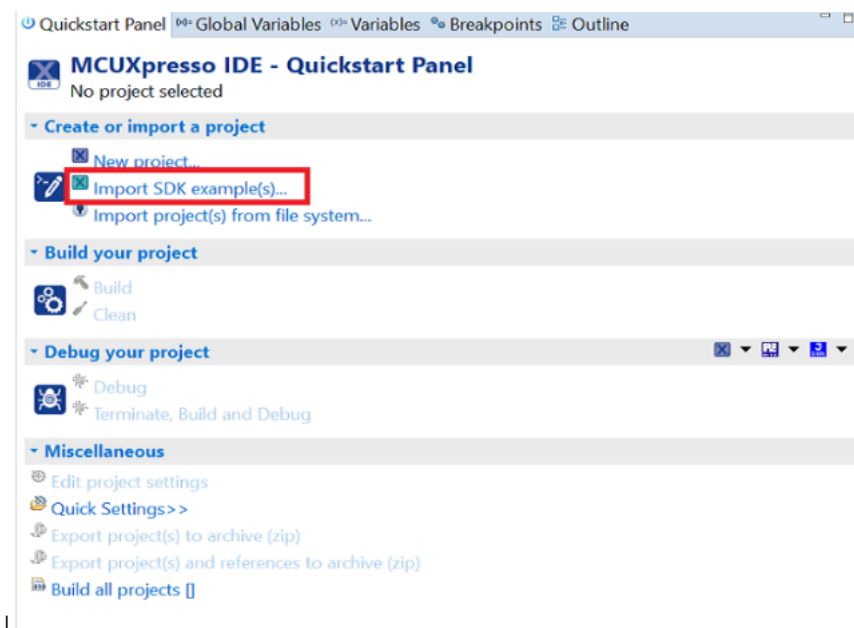
Parent topic: [Run a demo using MCUXpresso IDE](#)

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

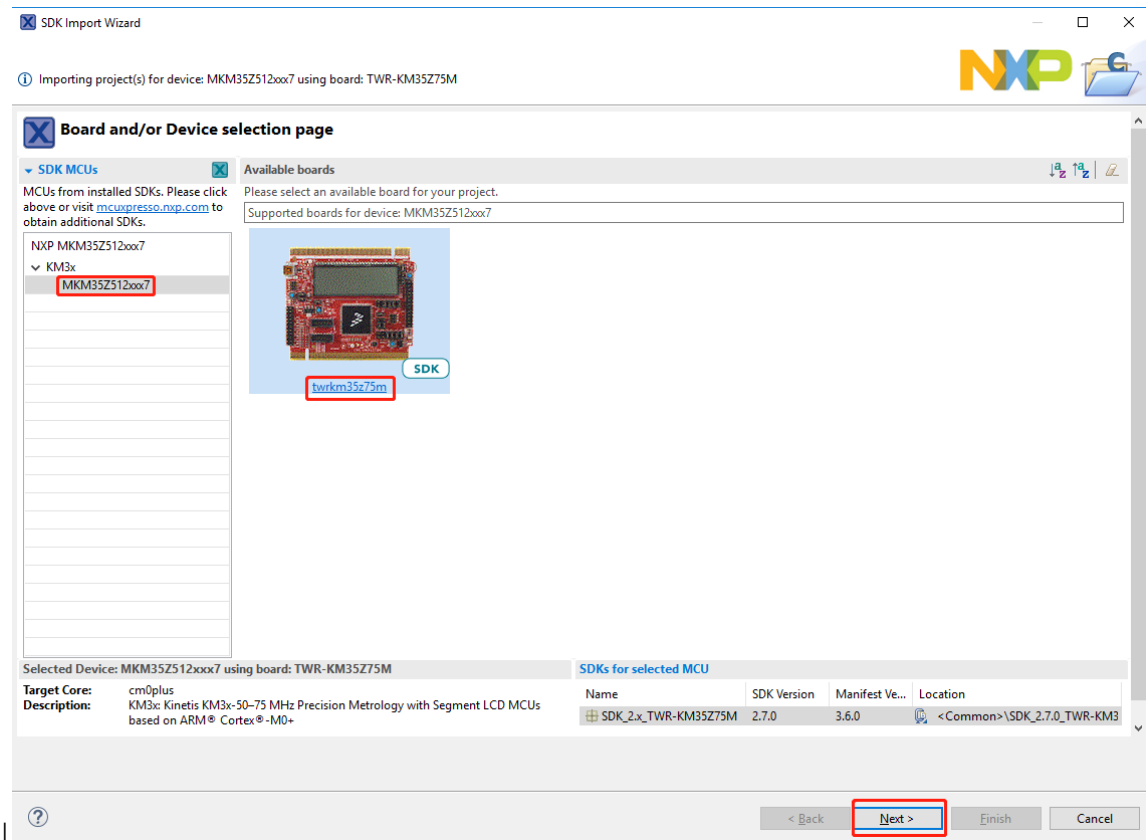
1. Drag and drop the SDK zip file into the **Installed SDKs** view to install an SDK. In the window that appears, click **OK** and wait until the import has finished.



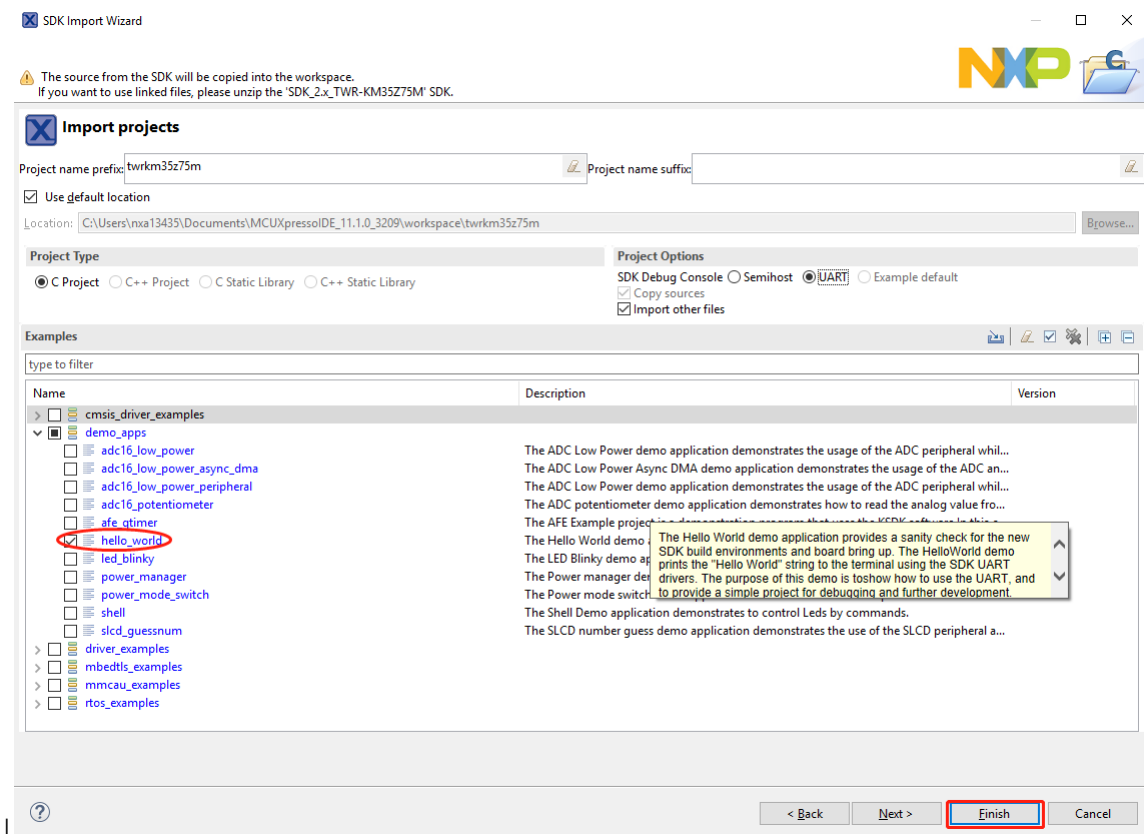
2. On the **Quickstart Panel**, click **Import SDK example(s)...**



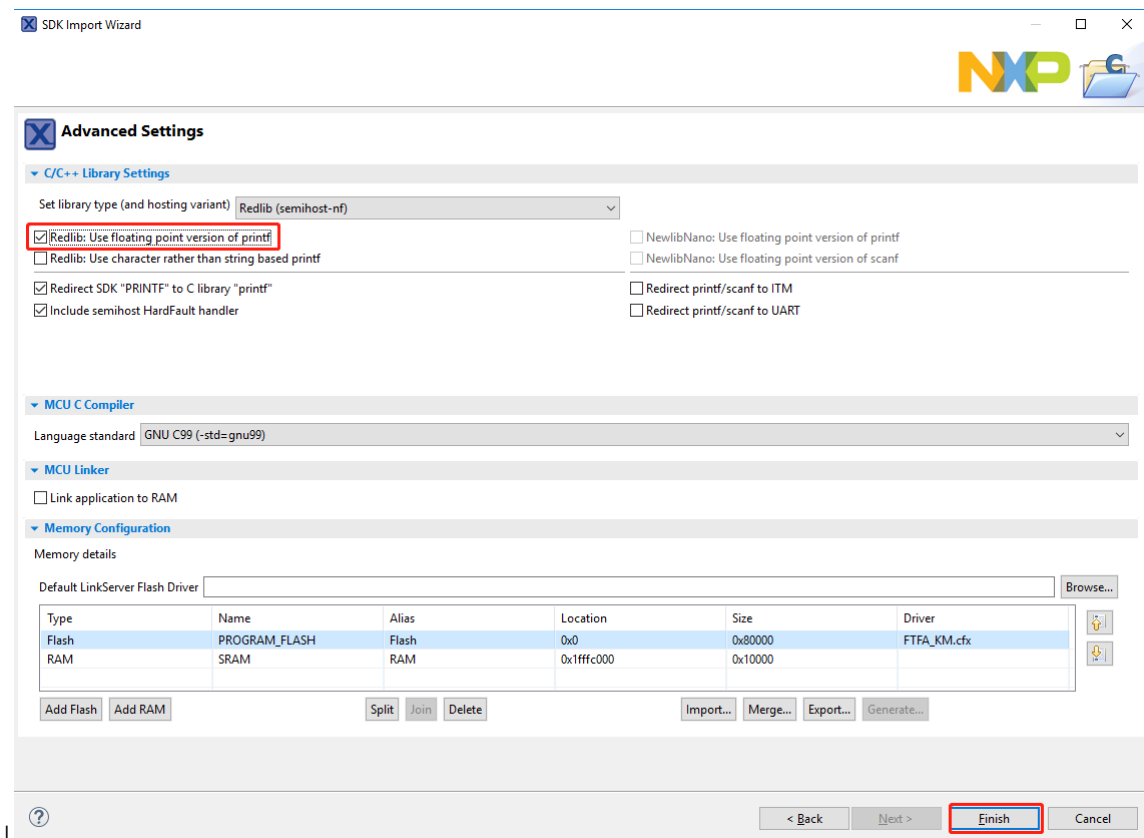
- In the window that appears, expand the **KM3x** folder and select **MKM35Z512xxx7**. Then, select **twrkm35z75m** and click **Next**.



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



5. Ensure the option of Redlib: Use floating point version of printf is selected if the cases' print floating-point numbers are on the terminal for demo applications such as adc_dma and adc_interrupt. Otherwise, it is not necessary to select this option. Then, click **Finish**.

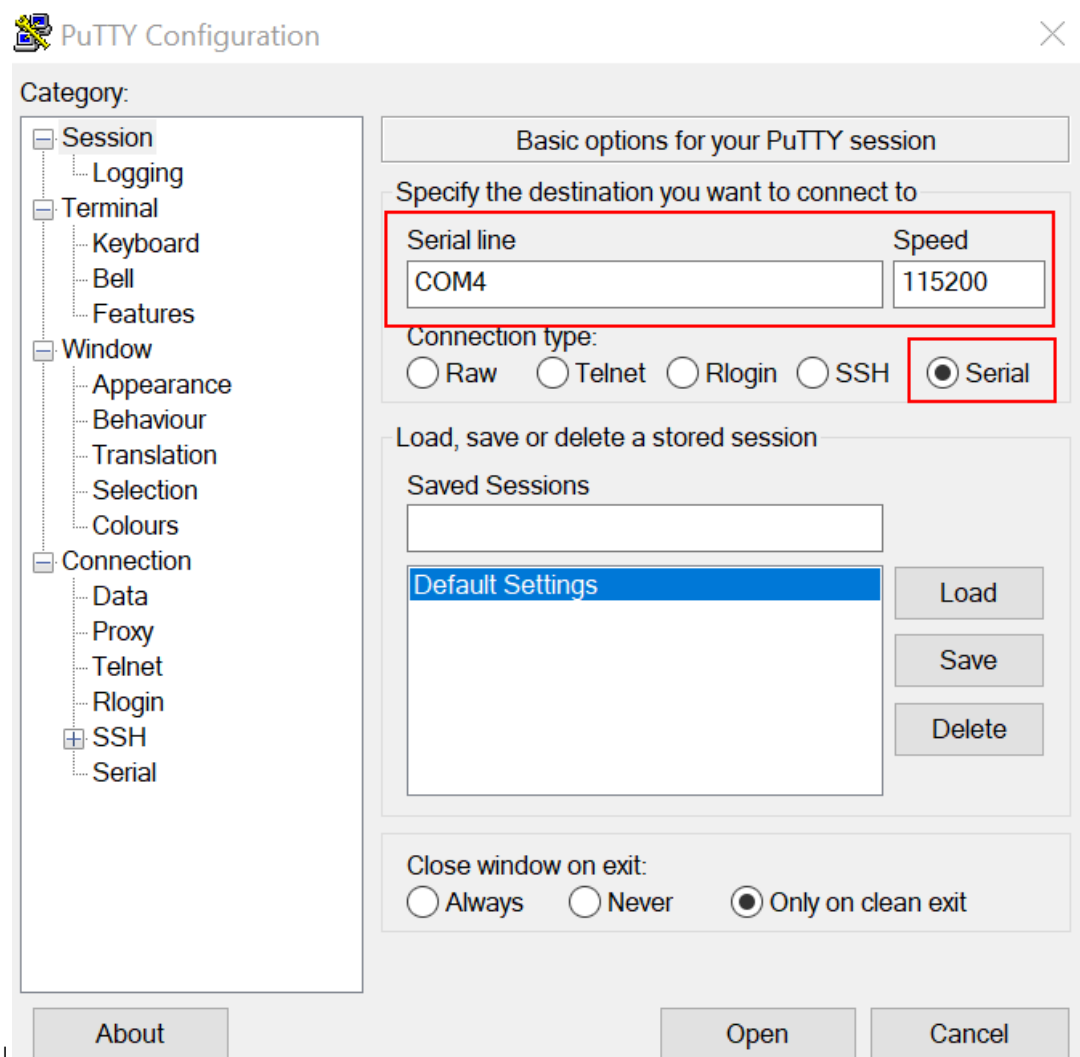


|
Parent topic: [Run a demo using MCUXpresso IDE](#)

Run an example application For more information on debug probe support in the MCUXpresso IDE 11.1.1, visit community.nxp.com.

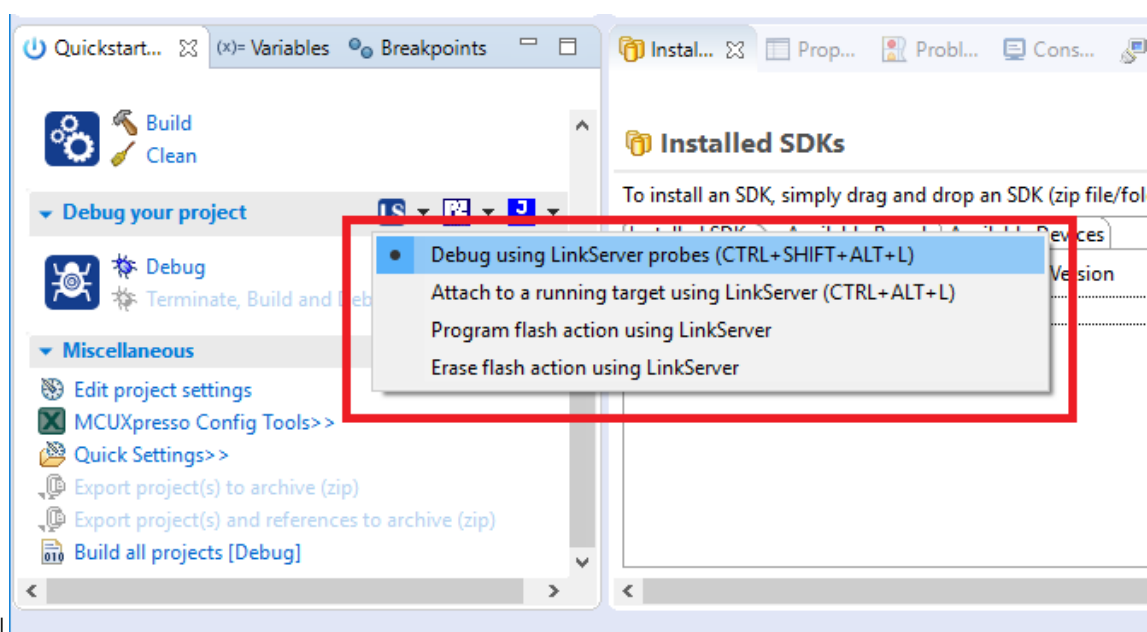
To download and run the application, perform these steps:

1. Reference the table in Appendix B to determine the debug interface that comes loaded on your specific hardware platform.
 - For boards with a P&E Micro interface, visit www.pemicro.com/support/downloads_find.cfm and download and install the P&E Micro Hardware Interface Drivers package.
 - If using J-Link with either a standalone debug pod or OpenSDA, install the J-Link software (drivers and utilities) from www.segger.com/jlink-software.html.
2. Connect the development platform to your PC via USB cable.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM port number, see Appendix A). Configure the terminal with these settings:
 1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)
 2. No parity
 3. 8 data bits



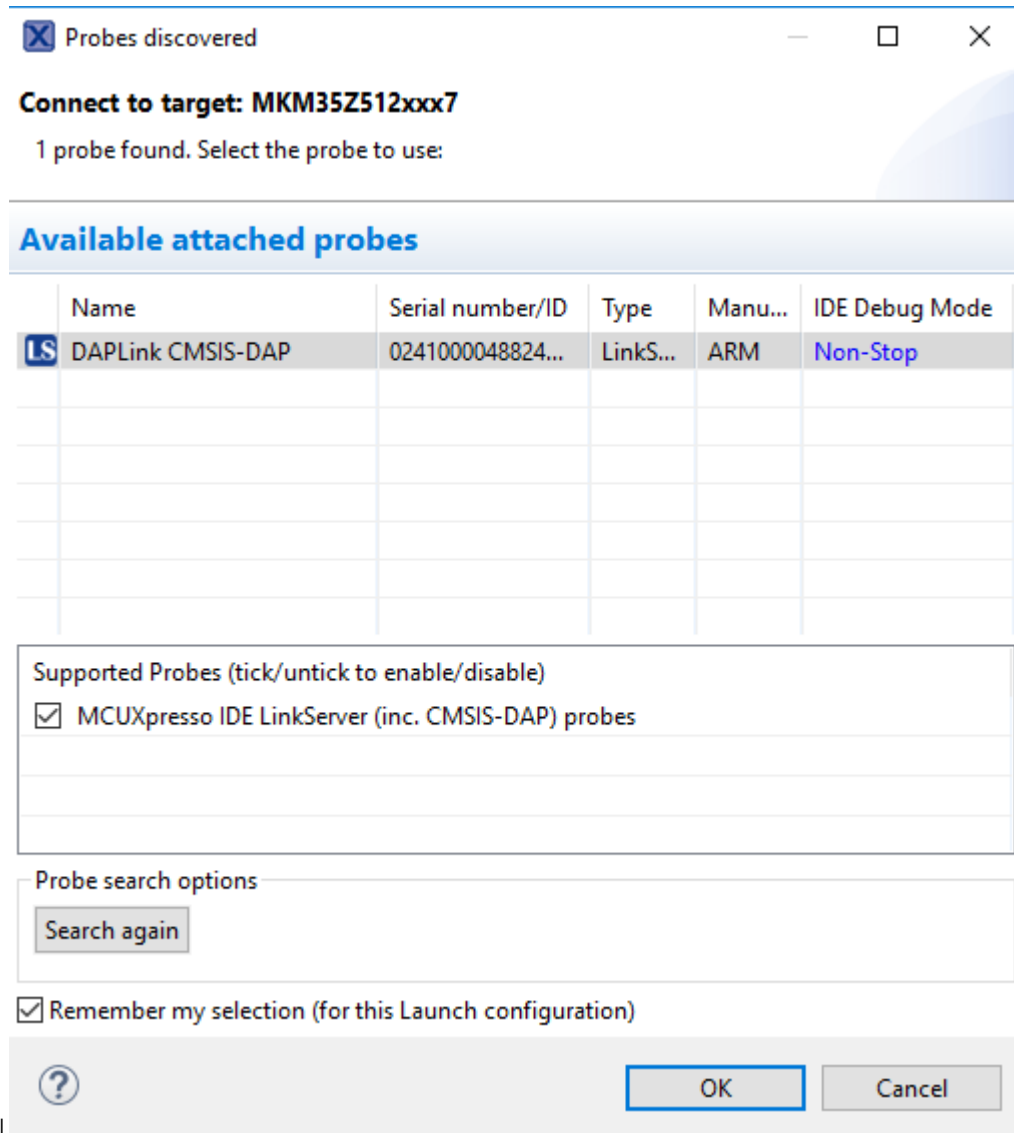
4. 1 stop bit |

4. On the **Quickstart Panel**, click **Debug** 'twrkm35z75m_demo_apps_hello_world' [Debug].

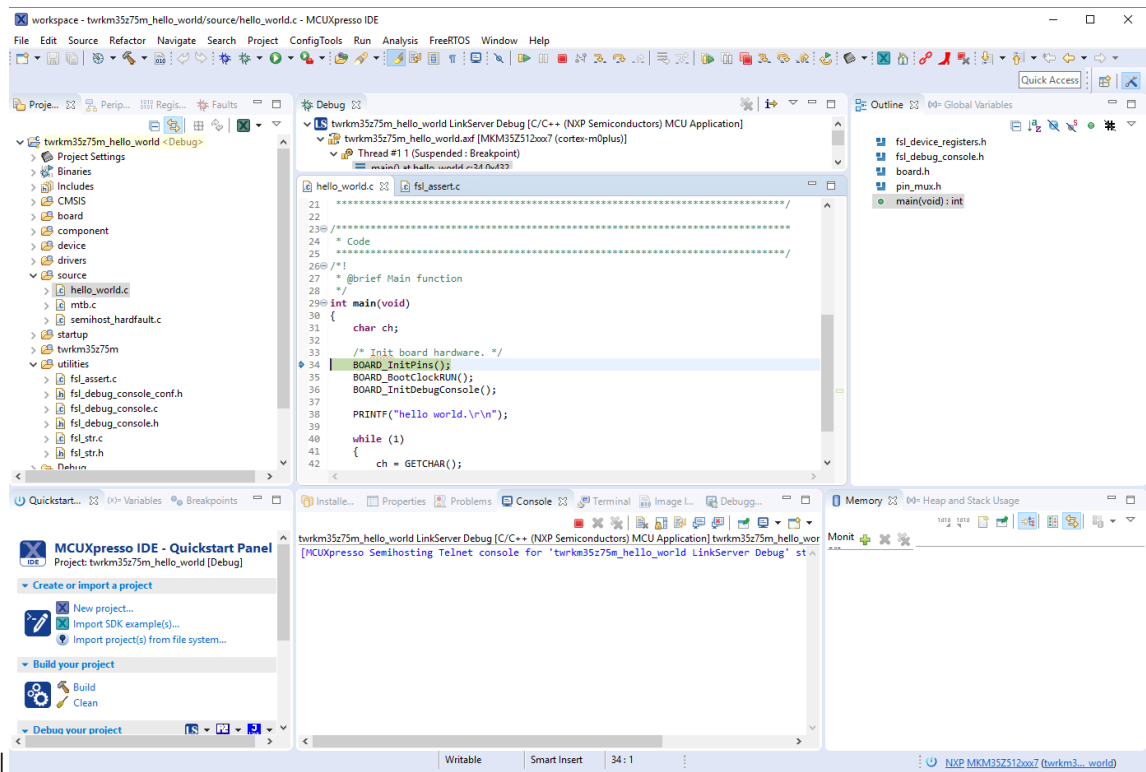


- The first time you debug a project, the **Debug Emulator Selection** dialog is displayed, showing all supported probes that are attached to your computer.

Select the probe through which you want to debug and click **OK**. (For any future debug sessions, the stored probe selection is automatically used, unless the probe cannot be found.)



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



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



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



Parent topic: [Run a demo using MCUXpresso IDE](#)

Run a demo application using IAR

This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK.

Note: IAR Embedded Workbench for Arm version is used as an example to show below steps, and the IAR toolchain should correspond to the latest supported version, as described in the *MCUXpresso SDK Release Notes for TWR-KM35Z75M* (document MCUXSDKKM35Z75MRN).

Build an example application Perform the following steps to build the `hello_world` example application.

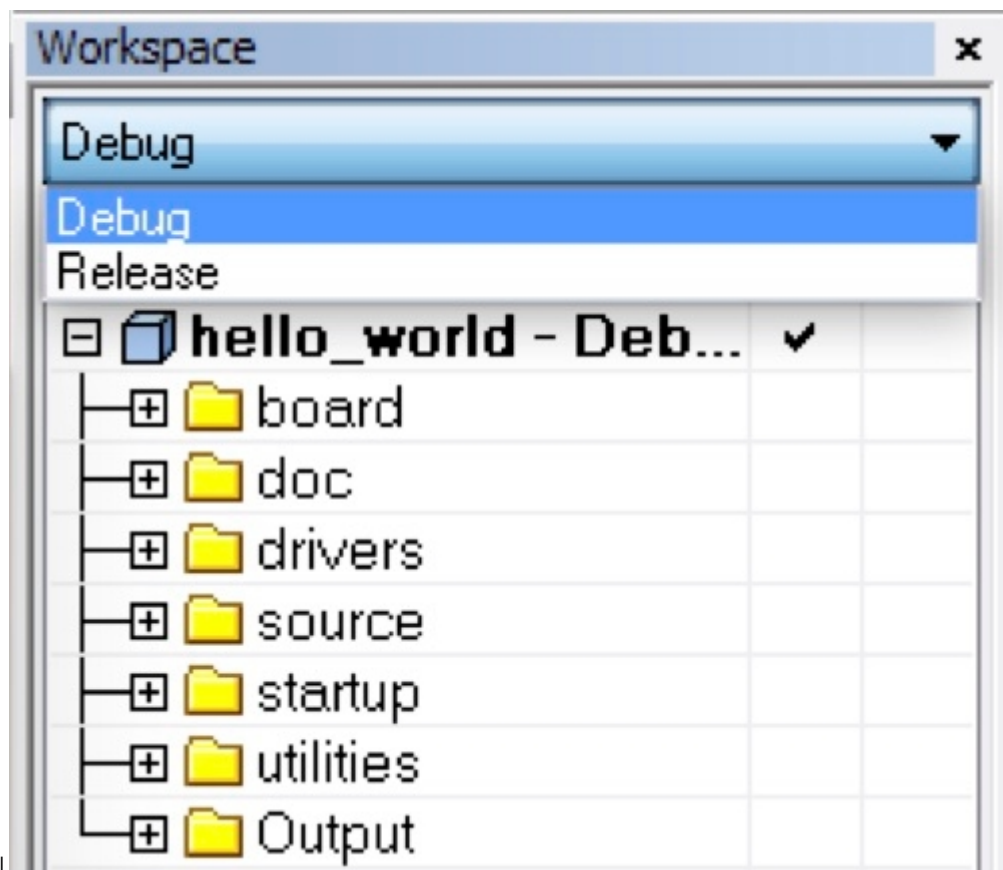
1. Open the desired demo application workspace. Most example application workspace files can be located using the following path:

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

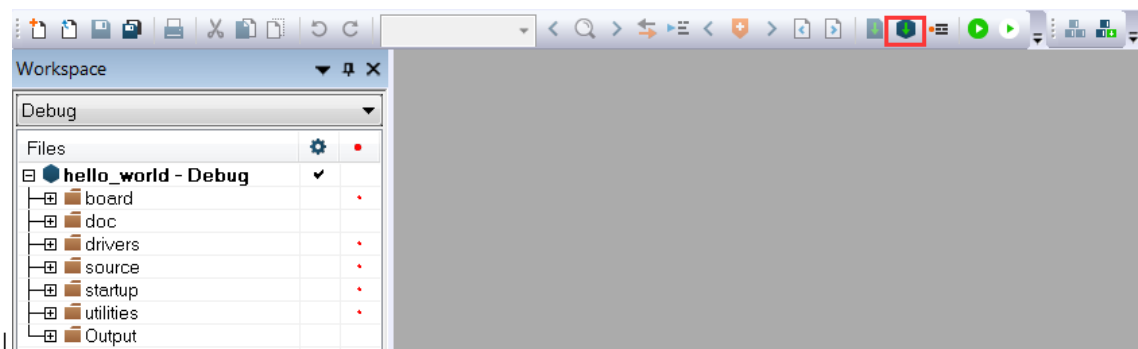
Using the TWR-KM35Z75M hardware platform as an example, the `hello_world` workspace is located in:

```
<install_dir>/boards/twrkm35z75m/demo_apps/hello_world/iar/hello_world.eww
```

2. Select the desired build target from the drop-down menu. For this example, select the **hello_world – debug** target.



3. To build the demo application, click the **Make** button highlighted in red below.

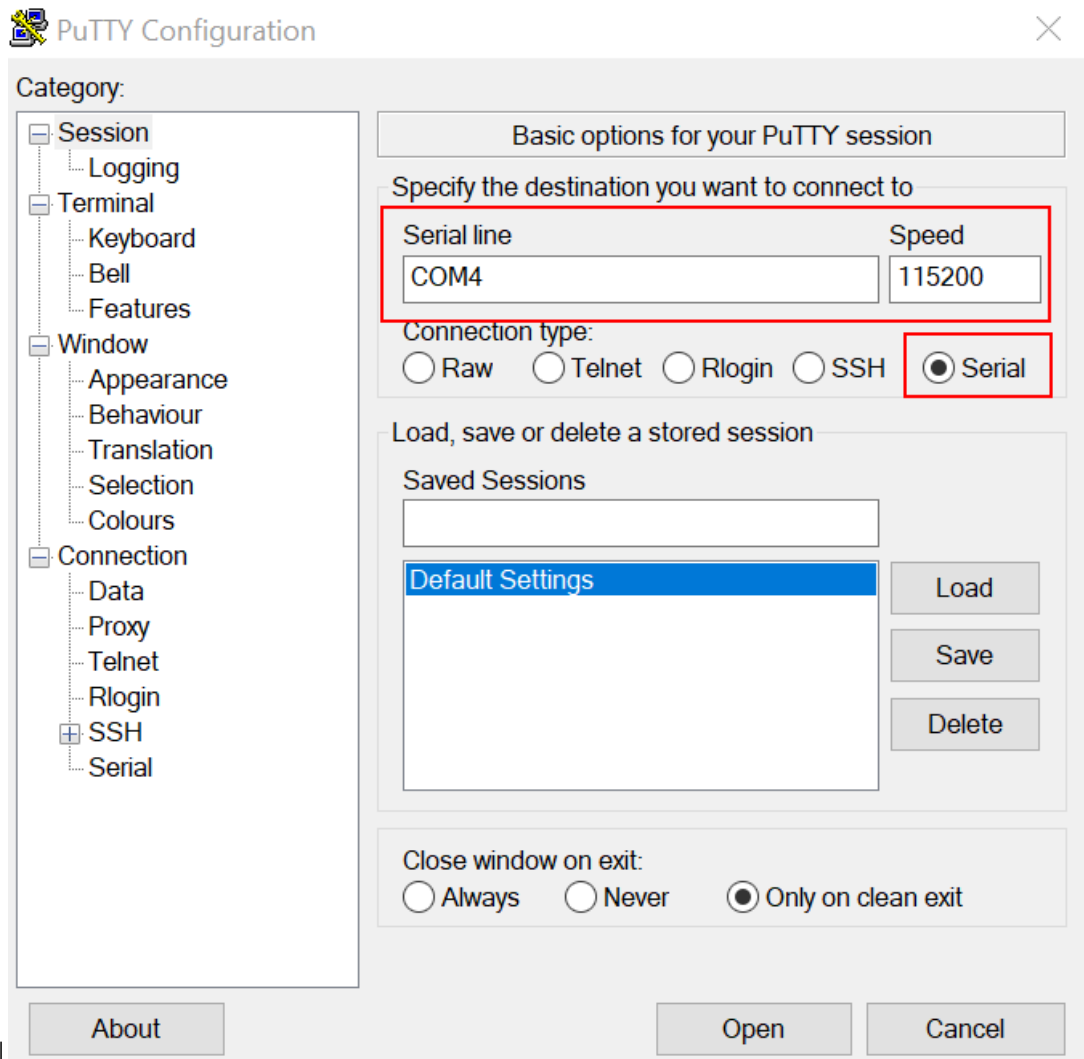


4. The build completes without errors.

Parent topic: [Run a demo application using IAR](#)

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

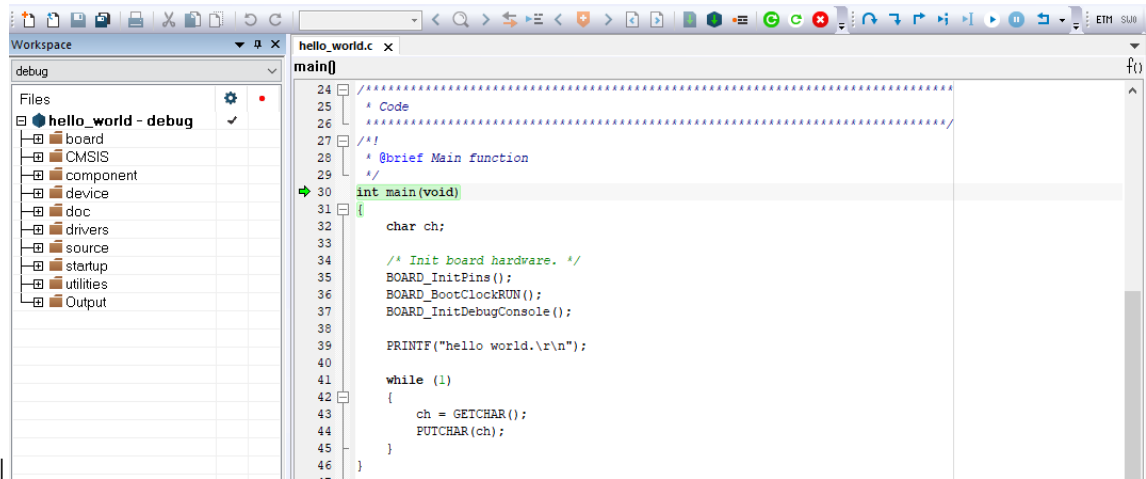
1. See the table in Default debug interfaces to determine the debug interface that comes loaded on your specific hardware platform.
 - For boards with CMSIS-DAP/mbed/DAPLink interfaces, visit developer.mbed.org/handbook/Windows-serial-configuration and follow the instructions to install the Windows operating system serial driver. If running on Linux OS, this step is not required.
 - For boards with P&E Micro interfaces, visit www.pemicro.com/support/downloads_find.cfm and download the P&E Micro Hardware Interface Drivers package.
2. Connect the development platform to your PC via USB cable.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug COM port (to determine the COM port number, see How to determine COM port). Configure the terminal with these settings:
 1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)
 2. No parity
 3. 8 data bits
 4. 1 stop bit



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



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



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



7. The `hello_world` application is now running and a banner is displayed on the terminal. If it does not appear, check your terminal settings and connections.



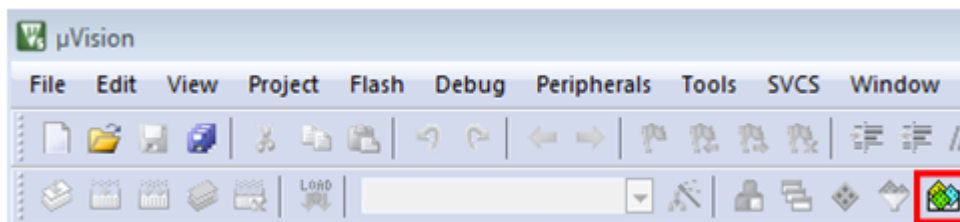
Parent topic: [Run a demo application using IAR](#)

Run a demo using Keil MDK/μVision

This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK. The `hello_world` demo application targeted for the TWR-KM35Z75M hardware platform is used as an example, although these steps can be applied to any demo or example application in the MCUXpresso SDK.

Install CMSIS device pack After the MDK tools are installed, Cortex Microcontroller Software Interface Standard (CMSIS) device packs must be installed to fully support the device from a debug perspective. These packs include things such as memory map information, register definitions, and flash programming algorithms. Follow these steps to install the appropriate CMSIS pack.

1. Open the MDK IDE, which is called μ Vision. In the IDE, select the **Pack Installer** icon.



2. After the installation finishes, close the Pack Installer window and return to the μ Vision IDE.

Parent topic: [Run a demo using Keil MDK/ \$\mu\$ Vision](#)

Build an example application

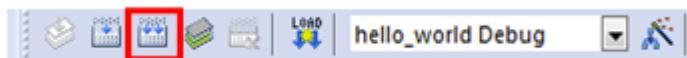
- Open the desired example application workspace in:

```
<install_dir>/boards/<board_name>/*<example\_type>*/<application_name>/mdk
```

The workspace file is named <demo_name>.uvmpw, so for this specific example, the actual path is:

```
<install_dir>/boards/twrkm35z75m/demo_apps/hello_world/mdk/hello_world.uvmpw
```

- To build the demo project, select the **Rebuild** button, highlighted in red.



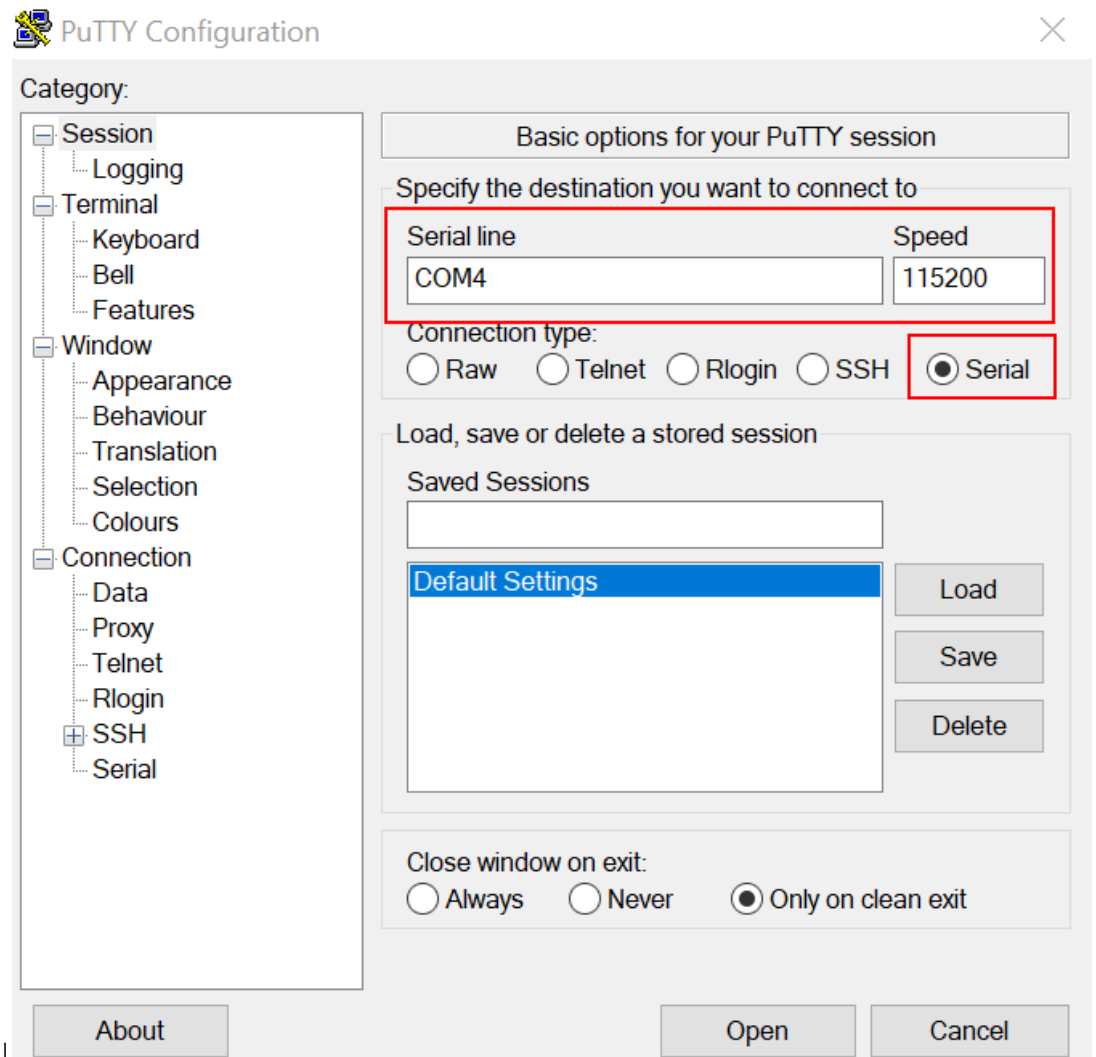
- The build completes without errors.

Parent topic: [Run a demo using Keil MDK/ \$\mu\$ Vision](#)

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

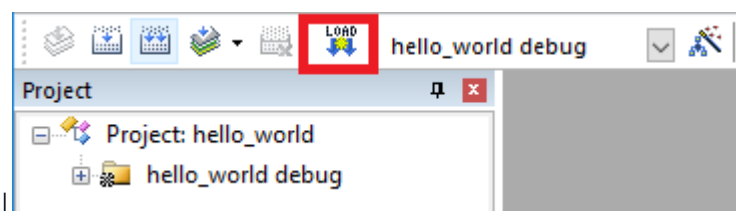
1. See the table in Default debug interfaces to determine the debug interface that comes loaded on your specific hardware platform.
 - For boards with the CMSIS-DAP/mbd/DAPLink interface, visit [mbed Windows serial configuration](#) and follow the instructions to install the Windows operating system serial driver. If running on Linux OS, this step is not required.
 - For boards with a P&E Micro interface, visit www.pemicro.com/support/downloads_find.cfm and download and install the P&E Micro Hardware Interface Drivers package.
 - If using J-Link either a standalone debug pod or OpenSDA, install the J-Link software (drivers and utilities) from www.segger.com/jlink-software.html.
2. Connect the development platform to your PC via USB cable using OpenSDA USB connector.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm and connect to the debug serial port number (to determine the COM port number, see How to determine COM port). Configure the terminal with these settings:
 1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)

2. No parity
3. 8 data bits

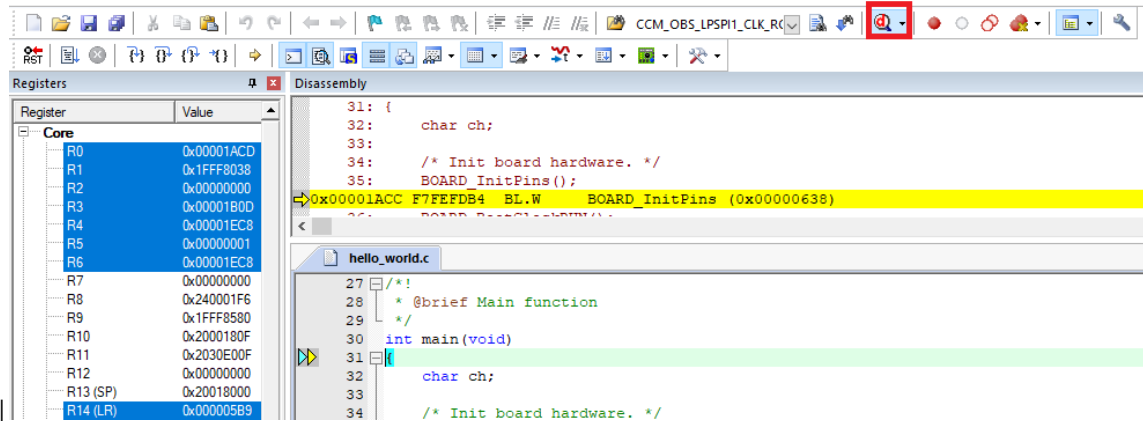


4. 1 stop bit

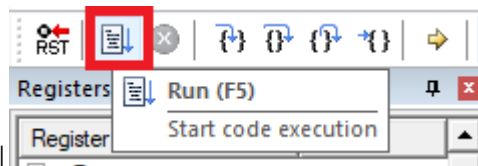
4. In μ Vision, after the application is built, click **Download** to download the application to the target.



5. After clicking **Download**, the application downloads to the target and is running. To debug the application, click **Start/Stop Debug Session**, highlighted in red.



6. Run the code by clicking **Run** to start the application.



The `hello\_world` application is now running and a banner is displayed on the terminal. If this does not appear, check your terminal settings and connections.

```

```

Parent topic: [Run a demo using Keil MDK/μVision](#)

Run a demo using ARMGCC / VSCODE

This section describes the steps to run an example application from the SDK archive using the ARMGCC / VSCODE toolchain.

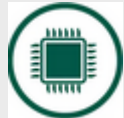
Refer to the [running a demo using MCUXpresso VSC](#) section for detailed instructions on setting up and configuring your project in Visual Studio Code.

Refer to the [CLI](#) section for detailed instructions on building and running your project from the command line.

MCUXpresso Config Tools

MCUXpresso Config Tools can help configure the processor and generate initialization code for the on chip peripherals. The tools are able to modify any existing example project, or create a new configuration for the selected board or processor. The generated code is designed to be used with MCUXpresso SDK version 2.x.

Table 1 describes the tools included in the MCUXpresso Config Tools.

Config Tool	Description	Image
Pins tool	For configuration of pin routing and pin electrical properties.	

| | **Clock tool** | For system clock configuration |



| | **Peripherals tools** | For configuration of other peripherals |



| | **TEE tool** | Configures access policies for memory area and peripherals helping to protect and

isolate sensitive parts of the application. |



| | **Device Configuration tool** | Configures Device Configuration Data (DCD) contained in the program image that the Boot ROM code interprets to set up various on-chip peripherals prior to the



program launch. |

MCUXpresso Config Tools can be accessed in the following products:

- **Integrated** in the MCUXpresso IDE. Config tools are integrated with both compiler and debugger which makes it the easiest way to begin the development.
- **Standalone version** available for download from www.nxp.com/mcuxpresso. Recommended for customers using IAR Embedded Workbench, Keil MDK μ Vision, or Arm GCC.
- **Online version** available on mcuxpresso.nxp.com. Recommended doing a quick evaluation of the processor or use the tool without installation.

Each version of the product contains a specific *Quick Start Guide* document MCUXpresso IDE Config Tools installation folder that can help start your work.

MCUXpresso IDE New Project Wizard

MCUXpresso IDE features a new project wizard. The wizard provides functionality for the user to create new projects from the installed SDKs (and from pre-installed part support). It offers user the flexibility to select and change multiple builds. The wizard also includes a library and provides source code options. The source code is organized as software components, categorized as drivers, utilities, and middleware.

To use the wizard, start the MCUXpresso IDE. This is located in the **QuickStart Panel** at the bottom left of the MCUXpresso IDE window. Select **New project**, as shown in *Figure 1*.



For more details and usage of new project wizard, see the *MCUXpresso_IDE_User_Guide.pdf* in the MCUXpresso IDE installation folder.

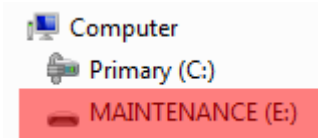
Updating OpenSDA firmware

Any NXP hardware platform that comes with an OpenSDA-compatible debug interface has the ability to update the OpenSDA firmware. This typically means to switch from the default application (either CMSIS-DAP/mbd/DAPLink or P&E Micro) to a SEGGER J-Link. This section contains the steps to switch the OpenSDA firmware to a J-Link interface. However, the steps can be applied to restoring the original image also. For reference, OpenSDA firmware files can be found at the links below:

- J-Link: Download appropriate image from www.segger.com/opensda.html. Choose the appropriate J-Link binary based on the table in Default debug interfaces. Any OpenSDA v1.0 interface should use the standard OpenSDA download (in other words, the one with no version). For OpenSDA 2.0 or 2.1, select the corresponding binary.
- CMSIS-DAP/mbd/DAPLink: DAPLink OpenSDA firmware is available at www.nxp.com/opensda.
- P&E Micro: Downloading P&E Micro OpenSDA firmware images requires registration with P&E Micro (www.pemicro.com).

Perform the following steps to update the OpenSDA firmware on your board for Windows and Linux OS users:

1. Unplug the board's USB cable.
2. Press the **Reset** button on the board. While still holding the button, plug the USB cable back into the board.
3. When the board re-enumerates, it shows up as a disk drive called **MAINTENANCE**.



4. Drag and drop the new firmware image onto the MAINTENANCE drive.

Note: If for any reason the firmware update fails, the board can always reenter maintenance mode by holding down **Reset** button and power cycling.

These steps show how to update the OpenSDA firmware on your board for Mac OS users.

1. Unplug the board's USB cable.
2. Press the **Reset** button of the board. While still holding the button, plug the USB cable back into the board.
3. For boards with OpenSDA v2.0 or v2.1, it shows up as a disk drive called **BOOTLOADER** in **Finder**. Boards with OpenSDA v1.0 may or may not show up depending on the bootloader version. If you see the drive in **Finder**, proceed to the next step. If you do not see the drive in **Finder**, use a PC with Windows OS 7 or an earlier version to either update the OpenSDA firmware, or update the OpenSDA bootloader to version 1.11 or later. The bootloader update instructions and image can be obtained from P&E Microcomputer website.
4. For OpenSDA v2.1 and OpenSDA v1.0 (with bootloader 1.11 or later) users, drag the new firmware image onto the BOOTLOADER drive in **Finder**.
5. For OpenSDA v2.0 users, type these commands in a Terminal window:

```
> sudo mount -u -w -o sync /Volumes/BOOTLOADER
> cp -X <path to update file> /Volumes/BOOTLOADER
```

Note: If for any reason the firmware update fails, the board can always reenter bootloader mode by holding down the **Reset** button and power cycling.

Revision history

This table summarizes revisions to this document.

Revision number	Date	Substantive changes
2.13.0	22 December 2022	Updated for MCUXpresso SDK v2.13.0
2.14.0	27 July 2023	Updated for MCUXpresso SDK v2.14.0

Legal information {#legal-conf-ApplicationnoteUsermanualUserguideReferencemanualNon-automotive}

1.3 Getting Started with MCUXpresso SDK GitHub

1.3.1 Getting Started with MCUXpresso SDK Repository

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

Overview

The GitHub Repository SDK approach offers:

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

Benefits of the Multi-Repository Approach

- **Scalability:** Easily add or update components without impacting the entire SDK.
- **Collaboration:** Enables distributed development across teams and repositories.
- **Version Control:** Independent versioning for components ensures better stability.
- **Automation:** Zephyr West simplifies dependency handling and repository synchronization.

Setup and Configuration

Follow these steps to prepare your development environment:

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

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

1. **Download the MCUXpresso Installer** from: [Dependency-Installation](#)
2. **Run the installer** and select “**MCUXpresso SDK Developer**” from the menu
3. **Click Install** and let it handle everything automatically

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

Essential Tools

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

Installation:

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

Setup:

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

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

Installation:

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

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

Installation:

```
pip install -U west
```

Minimum version: 1.2.0 or newer

Build System Tools

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

Recommended version: 3.30.0 or newer

Installation:

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

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

Minimum version: 1.12.1 or newer

Installation:

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

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

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

Installation: Follow the Ruby environment setup guide

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

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

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

Windows:

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

Linux/macOS:

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

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

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

Troubleshooting Installation Issues “Command not found” errors:

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

Python/pip issues:

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

Slow downloads:

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

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

Prerequisites Verify the requirements:

System Requirements:

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

Verification Commands:

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

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

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

Get the latest SDK from main branch:

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

Get SDK at specific revision:

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

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

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

```
cd mcuxpresso-sdk
```

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

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

```
west update
```

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

Best for:

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

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

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

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

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

Best for:

- Single board development

- Faster setup and reduced disk usage
- Focused development workflows

Examples:

```
# Update only repositories for FRDM-MCXW23 board
west update_board --set board frdm-mcxw23

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

Step 3: Verify Installation Confirm successful setup:

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

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

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

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

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

# List available repositories for the board before updating
west update_board --set board frdm-mcxw23 --list-repo
```

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

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

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

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

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

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

Benefits of Targeted Setup **Reduced Download Size**

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

- Typical reduction from 7 GB to 2GB

Optimized Workspace

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

Flexible Development

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

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

```
# Display repository information in console
west update_board --set board frdm-mcxw23 --list-repo

# Export repository information to YAML file for reference
west update_board --set board frdm-mcxw23 --list-repo -o board-repos.yml
```

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

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

```
# Generate board-specific SDK package
west update_board --set board frdm-mcxw23 -o frdm-mcxw23-sdk.zip
```

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

Workspace Management

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

For Complete SDK Workspace:

```
# Update manifest repository
cd manifests
git pull

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

For Targeted Workspace:

```
# Update manifest repository
cd manifests
git pull

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


Workspace Status Check workspace synchronization status:

```
# Show status of all repositories
west status
```

```
# Show detailed information about repositories
west list
```

Troubleshooting Network Issues:

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

Permission Issues:

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

Disk Space:

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

Repository Management Issues:

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

Next Steps With your workspace initialized:

1. Review [Workspace Structure](#) to understand the layout
2. Build your first project with [First Build Guide](#)
3. Explore [Development Workflows MCUXpresso VSCode](#) or [Development Workflows Command Line](#) for the details on project setup and execution

For advanced repository management, see the [west tool documentation](#).

Explore SDK Structure and Content

Learn about the organization of the SDK and its components:

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

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

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

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

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

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

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

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

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

- [nxp-mcuxpresso](#)
- [NXP](#)
- [nxp-zephyr](#)

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

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

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

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

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

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

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

Top-Level Organization

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

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

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

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

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

Common Example Files

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

Board-Specific Files

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

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

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

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

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

Documentation Structure SDK documentation is distributed across multiple locations:

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

For complete documentation, refer to the [online documentation](#).

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

1. General README: `examples/demo_apps/hello_world/readme.md`

- What the example does
- General functionality description
- Common usage information

2. Board-Specific README: `examples/_boards/{board_name}/demo_apps/hello_world/example_board_readme.md`

- Board-specific setup instructions
- Hardware connections required
- Board-specific behavior notes

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

Development Workflows

Get started with building and running projects:

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

Prerequisites

- GitHub Repository SDK workspace initialized OR Repository-Layout SDK Package extracted
- Development board connected via USB
- Build tools installed per [Installation Guide](#)

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

```
west list _project -p examples/demo_apps/hello_world
```

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

```
west list _project -p examples/demo_apps/hello_world -t armgcc
```

Basic Build Process

Simple Build Build the hello_world example with default settings:

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

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

Specifying Configuration

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

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

Alternative Toolchains

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

# Other toolchains as supported by the example
```

Multicore Applications

 For multicore devices, specify the core ID:

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

For multicore projects using sysbuild:

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

Flash an Application

 Flash the built application to your board:

```
west flash -r linkserver
```

Debug

 Start a debug session:

```
west debug -r linkserver
```

Common Build Options

Clean Build

 Force a complete rebuild:

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

Dry Run

 See the commands that get executed without running them:

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

Device Variants

 For boards supporting multiple device variants:

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

Project Configuration

CMake Configuration Only Run configuration without building:

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

Interactive Configuration Launch the configuration GUI:

```
west build -t guiconfig
```

Troubleshooting

Build Failures Use pristine builds to resolve dependency issues:

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

Getting Help View the help information for west build:

```
west build -h
```

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

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

Next Steps

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

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

Prerequisites

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

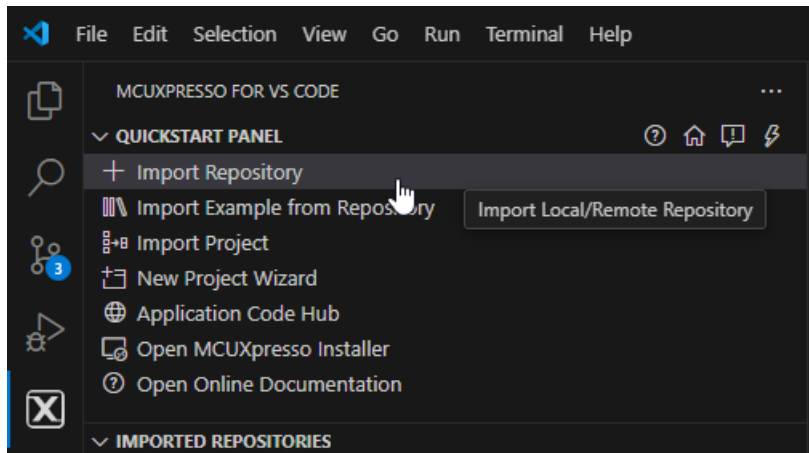
Extension Installation

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

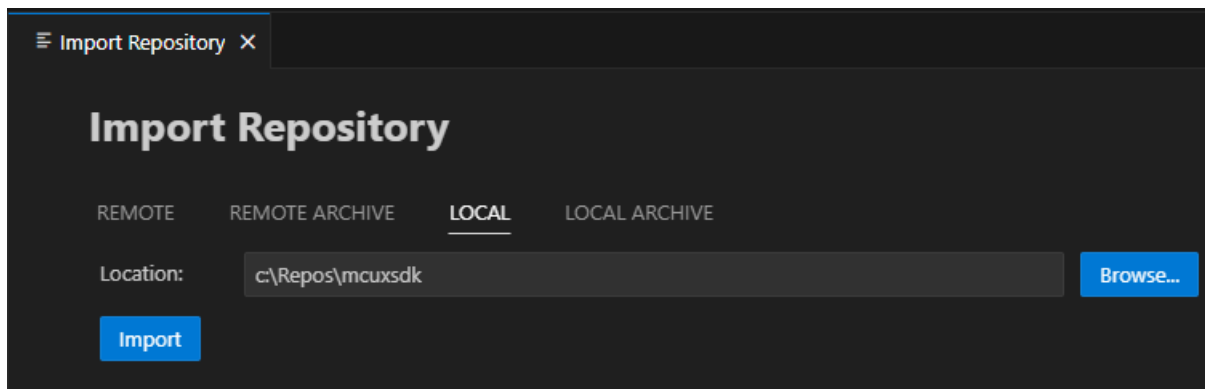
SDK Import and Setup

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

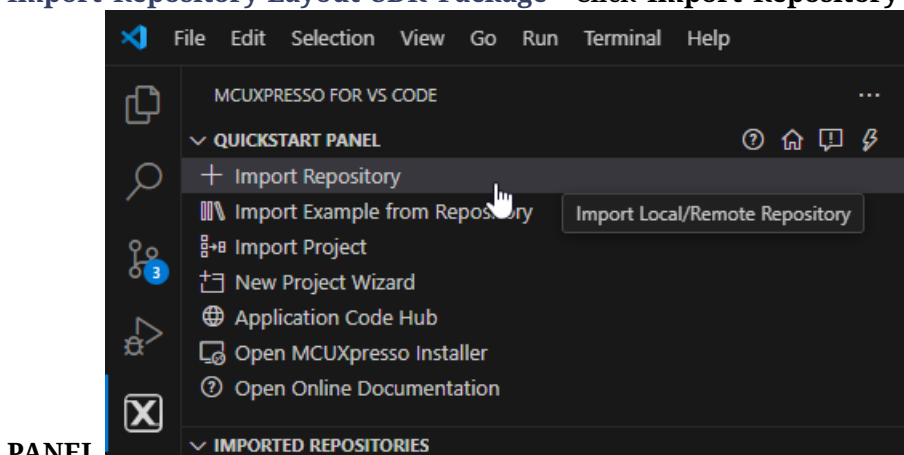
Import GitHub Repository SDK Click **Import Repository** from the **QUICKSTART PANEL**



Note: You can import the SDK in several ways. Refer to [MCUXpresso for VS Code Wiki](#) for details. Select **Local** if you've already obtained the SDK according to [setting up the repo](#). Select your location and click **Import**.

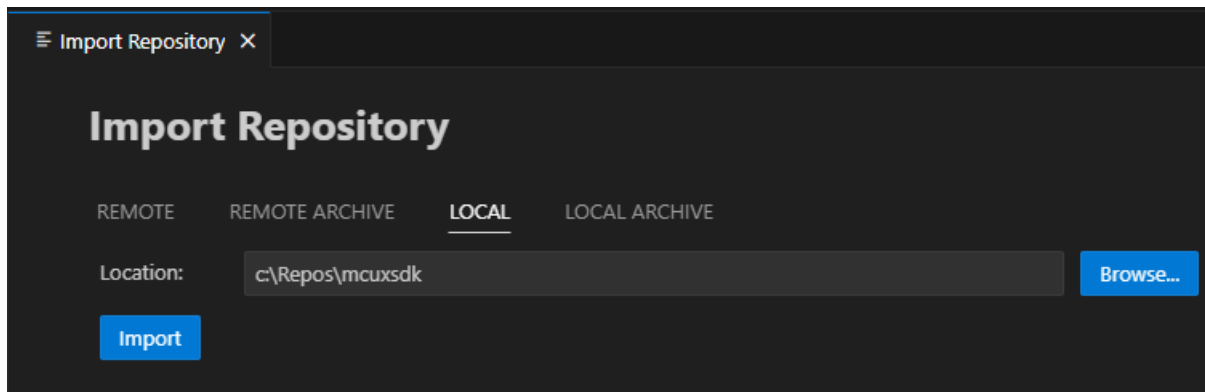


Import Repository-Layout SDK Package Click **Import Repository** from the **QUICKSTART**

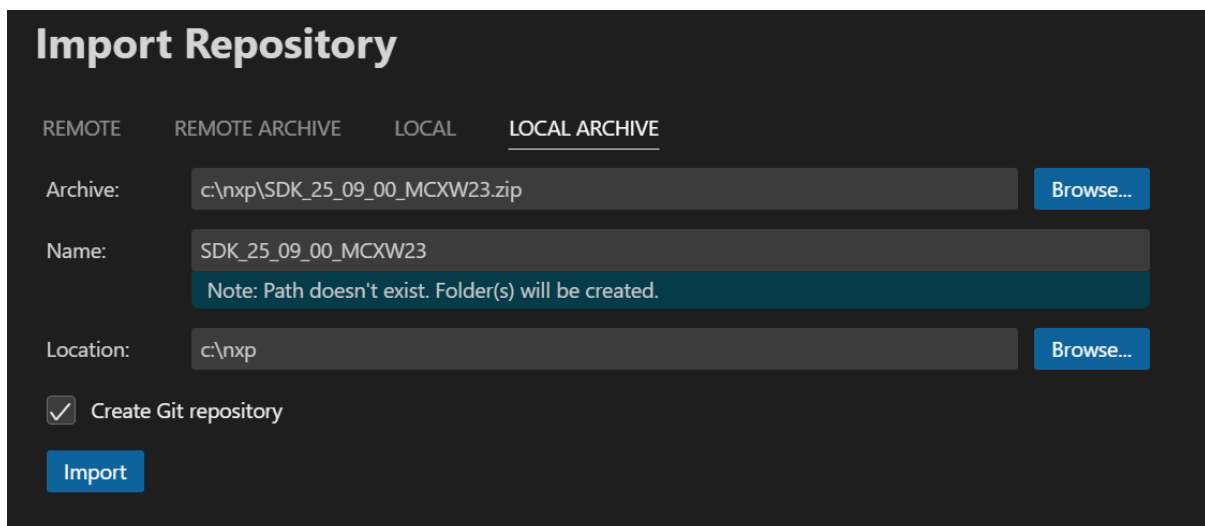


PANEL

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



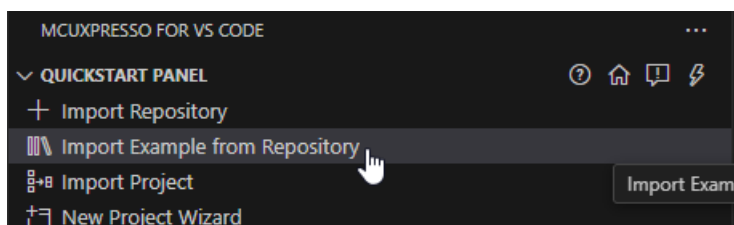
Else if the SDK is ZIP archive, select **Local Archive**, browse to the downloaded SDK ZIP file, fill the link of expect location, then click **Import**.



Building Example Applications

Import Example Project

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



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

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

3. Click **Import**

Import Example from Repository

Import Example from Repository

Repository:
c:\Repos\mcuxsdk (MCUXpresso SDK Repository)

Toolchain:
(Arm GNU Toolchain 13.2.rel1 (Build arm-13.7)) 13.2.1 20231009 (C:\NXP\MCUXpressoIDE_24.

Board:
FRDM-MCXC444


FRDM-MCXC444

Template:
demo_apps/hello_world

The HelloWorld demo prints the "Hello World" string to the terminal using the SDK UART drivers and repeat what user input. The purpose of this demo is to show how to use the UART, and to provide a simple project for debugging and further development.
Please refer to [README](#) file for more details.

App type:
Freestanding application

Name:
frdmxc444_hello_world

Location:
c:\npx_examples
Browse...

Note: Path doesn't exist. Folder(s) will be created.

☐ Open readme file after project is imported

Import

Application Types **Repository Applications:**

- Located inside the MCUXpresso SDK
- Integrated with SDK workspace

Freestanding Applications:

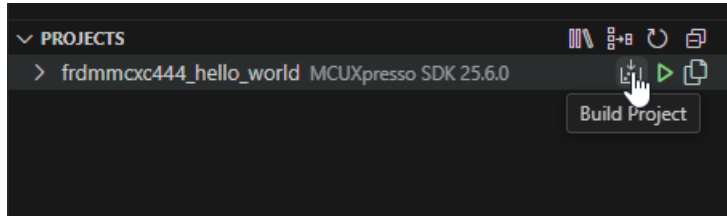
- Imported to user-defined location
- Independent of SDK location

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

Building Projects

Build Process

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

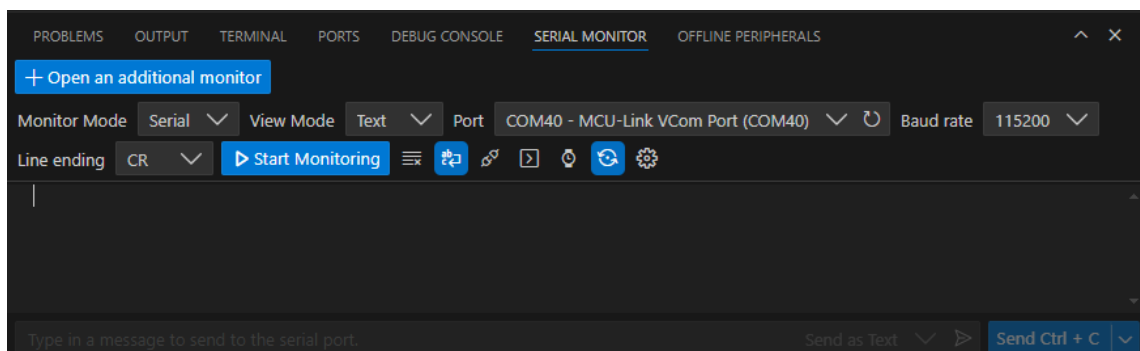


The integrated terminal will display build output at the bottom of the VS Code window.

Running and Debugging

Serial Monitor Setup

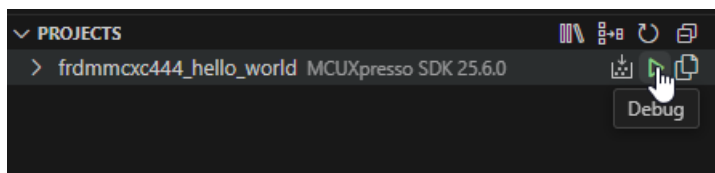
1. Open **Serial Monitor** from VS Code's integrated terminal



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

Debug Session

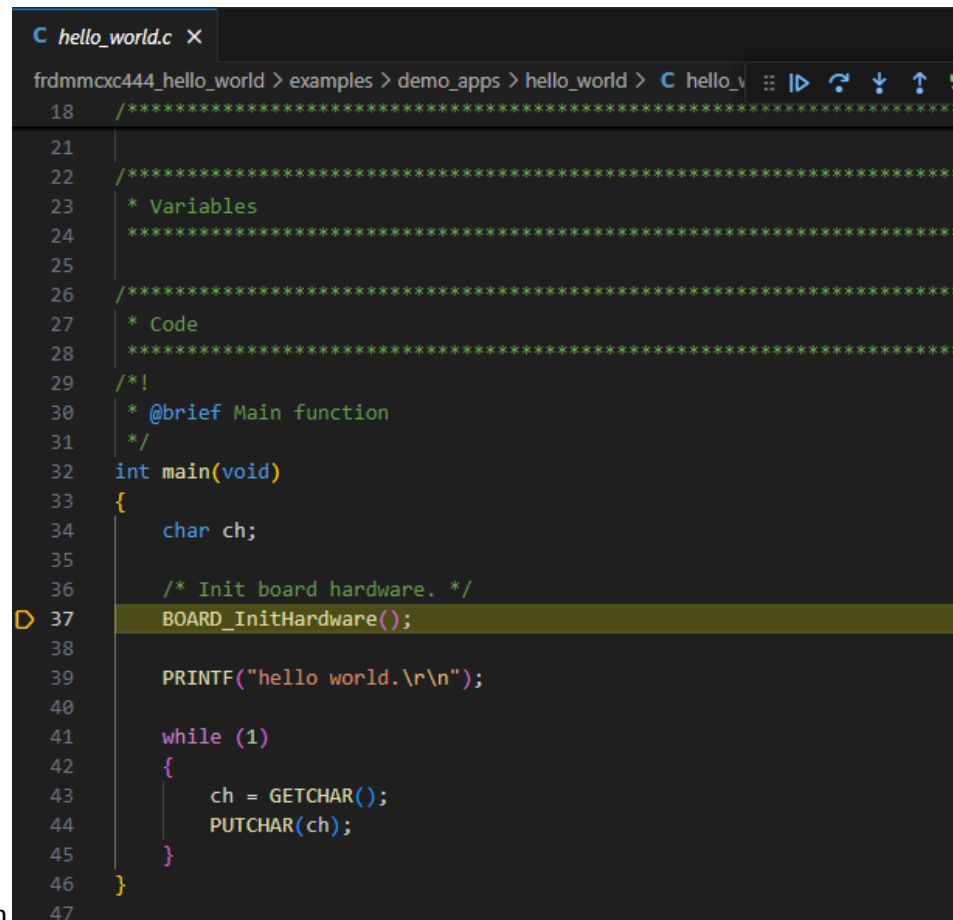
1. Navigate to **PROJECTS** view
2. Click the play button to initiate a debug session



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

Debug Controls Use the debug controls to manage execution:

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



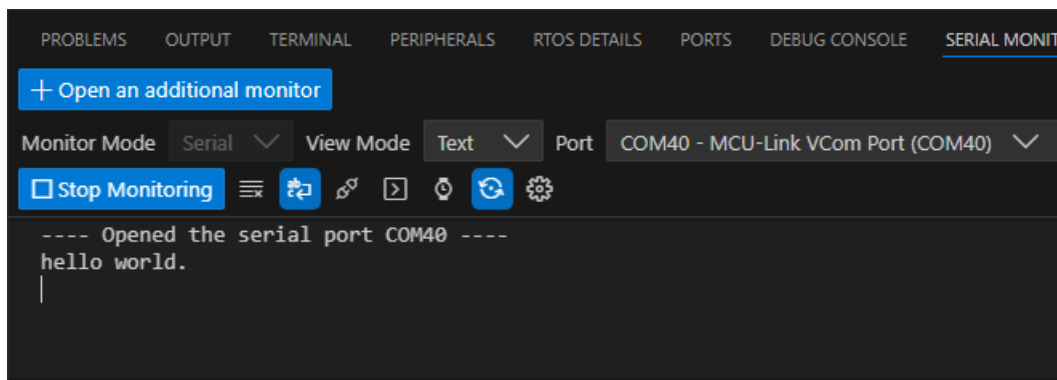
```

C hello_world.c X
frdm-mxc444_hello_world > examples > demo_apps > hello_world > C hello_w
18  /*****
21
22  /*****
23  * Variables
24  *****/
25
26  /*****
27  * Code
28  *****/
29  /*!
30  * @brief Main function
31  */
32  int main(void)
33  {
34      char ch;
35
36      /* Init board hardware. */
37      BOARD_InitHardware();
38
39      PRINTF("hello world.\r\n");
40
41      while (1)
42      {
43          ch = GETCHAR();
44          PUTCHAR(ch);
45      }
46  }
47

```

- **Stop:** Terminate debug session

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



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

Project Configuration

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

- Example applications
- Board configurations

- Middleware components
- Build system integration

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

Troubleshooting

Import Issues SDK not detected:

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

Project import failures:

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

Build Problems Build failures:

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

Debug Issues Debug session fails:

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

Serial monitor problems:

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

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

Advanced Features

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

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

Next Steps

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

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

Prerequisites

- GitHub Repository SDK workspace initialized OR Repository-Layout SDK Package extracted
- Development tools installed per [Installation Guide](#)
- Target board connected via USB

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

```
west list _project -p examples/demo_apps/hello_world
```

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

```
west list _project -p examples/demo_apps/hello_world -t armgcc
```

Basic Build Commands

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

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

Specifying Build Configuration

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

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

Multicore Applications For multicore devices, specify the core ID:

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

For multicore projects using sysbuild:

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

Shield Support For boards with shields:

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

Advanced Build Options

Clean Builds Force a complete rebuild:

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

Dry Run See what commands would be executed:

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

Device Variants For boards supporting multiple device variants:

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

Project Configuration

CMake Configuration Only Run configuration without building:

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

Interactive Configuration Launch the configuration GUI:

```
west build -t guiconfig
```

Flashing and Debugging

Flash Application Flash the built application to your board:

```
west flash -r linkserver
```

Debug Session Start a debugging session:

```
west debug -r linkserver
```

IDE Project Generation Generate IDE project files for traditional IDEs:

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

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

Note: Ruby installation is required for IDE project generation. See [Installation Guide](#) for setup instructions.

Troubleshooting

Build Failures Use pristine builds to resolve dependency issues:

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

Toolchain Issues Verify environment variables are set correctly:

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

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

Getting Help Display help information:

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

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

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

Best Practices

Project Organization

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

Build Efficiency

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

Development Workflow

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

Next Steps

- Explore [VS Code Development](#) for integrated development experience
- Review [Workspace Structure](#) to understand SDK organization
- Refer build system documentation for advanced configurations

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

Prerequisites

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

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

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

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

Manual Workflow Use the following steps:

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

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

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

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

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

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

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

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

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

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

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

1.4 Release Notes

1.4.1 MCUXpresso SDK Release Notes

Overview

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

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

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

MCUXpresso SDK

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

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

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

Development tools

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

- MCUXpresso IDE, Rev. 25.06.xx
- IAR Embedded Workbench for Arm, version is 9.60.4
- Keil MDK, version is 5.42
- MCUXpresso for VS Code v25.09
- GCC Arm Embedded Toolchain 14.2.x

Supported development systems

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

Devel- opment boards	MCU devices
TWR-KM35Z75M	MKM35Z256VLL7, MKM35Z256VLL7R, MKM35Z256VLQ7, MKM35Z256VLQ7R, MKM35Z512VLL7, MKM35Z512VLL7R, MKM35Z512VLQ7 , MKM35Z512VLQ7R

MCUXpresso SDK release package

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

Device support The device folder contains the whole software enablement available for the specific System-on-Chip (SoC) subfamily. This folder includes clock-specific implementation, device register header files, device register feature header files, and the system configuration source files. Included with the standard SoC support are folders containing peripheral drivers, toolchain support, and a standard debug console. The device-specific header files provide a direct access to the microcontroller peripheral registers. The device header file provides an overall SoC memory mapped register definition. The folder also includes the feature header file for each peripheral on the microcontroller. The toolchain folder contains the startup code and linker files for each supported toolchain. The startup code efficiently transfers the code execution to the main() function.

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

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

RTOS

FreeRTOS Real-time operating system for microcontrollers from Amazon

Middleware

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

MCU Boot MCU Boot (formerly KBOOT) NXP/Freescale proprietary loader

TinyCBOR Concise Binary Object Representation (CBOR) Library

PKCS#11 The PKCS#11 standard specifies an application programming interface (API), called “Cryptoki,” for devices that hold cryptographic information and perform cryptographic functions. Cryptoki follows a simple object based approach, addressing the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices), presenting to applications a common, logical view of the device called a “cryptographic token”.

MMCAU The NXP Memory-Mapped Cryptographic Acceleration Unit

mbedtls mbedtls SSL/TLS library v2.x

Metering Metering Library

llhttp HTTP parser llhttp

FreeMASTER FreeMASTER communication driver for 32-bit platforms.

Release contents

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

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

Known issues

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

Cannot add SDK components into FreeRTOS projects

It is not possible to add any SDK components into FreeRTOS project using the MCUXpresso IDE New Project wizard.

1.5 ChangeLog

1.5.1 MCUXpresso SDK Changelog

Board Support Files

board

[25.06.00]

- Initial version

clock_config

[25.06.00]

- Initial version

pin_mux

[25.06.00]

- Initial version
-

ADC16

[2.3.0]

- Improvements
 - Added new API ADC16_EnableAsynchronousClockOutput() to enable/disable ADACK output.
 - In ADC16_GetDefaultConfig(), set enableAsynchronousClock to false.

[2.2.0]

- Improvements
 - Added hardware average mode in adc_config_t structure, then the hardware average mode can be set by invoking ADC16_Init() function.

[2.1.0]

- New Features:
 - Supported KM series' new ADC reference voltage source, bandgap from PMC.

[2.0.3]

- Bug Fixes
 - Fixed IAR warning Pa082: the order of volatile access should be defined.

[2.0.2]

- Improvements
 - Used conversion control feature macro instead of that in IO map.

[2.0.1]

- Bug Fixes
 - Fixed MISRA-2012 rules.
 - * Rule 16.4, 10.1, 13.2, 14.4 and 17.7.

[2.0.0]

- Initial version
-

AFE**[2.0.3]**

- Improvements
 - Fixed CERT-C issues

[2.0.2]

- Bug Fixes
 - Fixed MISRA C-2012 rule 10.1, rule 10.4 and so on.

[2.0.1]

- Improvements
 - Changed type modifiers from `const xx_Type * s_xxBases` to `xx_Type *const s_xxBases`.
 - Added static modifier for `s_xxx` variables defined in drivers.

[2.0.0]

- Initial version.
-

CLOCK**[2.2.3]**

- Bug Fixes
 - Fixed an issue that in `CLOCK_SetFbeMode()` C4 register not updated.

[2.2.2]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 18.1.

[2.2.1]

- Bug Fixes
 - Fixed MISRA C-2012 rule 10.1, rule 10.4, rule 14.4 and so on.

[2.2.0]

- Initial version.
-

CMP

[2.0.3]

- Improvements
 - Updated to clear CMP settings in DeInit function.

[2.0.2]

- Bug Fixes
 - Fixed the violations of MISRA 2012 rules:
 - * Rule 10.3

[2.0.1]

- Bug Fixes
 - Fixed MISRA-2012 rules.
 - * Rule 14.4, rule 10.3, rule 10.1, rule 10.4 and rule 17.7.

[2.0.0]

- Initial version.
-

COMMON

[2.6.3]

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

[2.6.2]

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

[2.6.1]

- Improvements
 - Support Cortex M23.

[2.6.0]

- Bug Fixes
 - Fix CERT-C violations.

[2.5.0]

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

[2.4.3]

- Improvements
 - Enable irq's that mount under `irqsteer` interrupt extender.

[2.4.2]

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

[2.4.1]

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

[2.4.0]

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

[2.3.3]

- New Features
 - Added `NETC` into status group.

[2.3.2]

- Improvements
 - Make driver `aarch64` compatible

[2.3.1]

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

[2.3.0]

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

[2.2.10]

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

[2.2.9]

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

[2.2.8]

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

[2.2.7]

- Other Change
 - Added MECC status group definition.

[2.2.6]

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

[2.2.5]

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

[2.2.4]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.4]

- New Features
 - Added OTFAD into status group.

[2.1.3]

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

[2.1.2]

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

[2.1.1]

- Bug Fixes
 - Deleted and optimized repeated macro.

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

CRC

[2.0.5]

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

[2.0.4]

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

[2.0.3]

- Bug fix:
 - Fix MISRA issues.

[2.0.2]

- Bug fix:
 - Fix MISRA issues.

[2.0.1]

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

[2.0.0]

- Initial version.
-

DMA

[2.1.3]

- Bug Fixes
 - Fixed coverity issues with CERT INT30-C, CERT INT31-C compliance.

[2.1.2]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.3.

[2.1.1]

- Improvements
 - Corrected the dma channel feature macro from FSL_FEATURE_DMAMUX_MODULE_CHANNEL to FSL_FEATURE_DMA_MODULE_CHANNEL.

[2.1.0]

- Improvements
 - Added api DMA_PrepareTransferConfig to expose option address increment.
 - Added api DMA_EnableAutoStopRequest to support auto stop request feature.

[2.0.2]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.4, 10.3, 14.4, 16.4, 11.6, 10.1.

[2.0.1]

- Bug Fixes
 - By adding parenthesis, fixed the build fail of DMA driver due to rule 12.5, MISRA C 2004.

[2.0.0]

- Initial version.
-

DMAMUX

[2.1.3]

- Improvements
 - Wrap DMAMUX_GetInstance into FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL to avoid build issues.

[2.1.2]

- Bug Fixes
 - Add macro `FSL_DMAMUX_CHANNEL_NUM` to calculate correct DMAMUX channel number when input EDAM channel number.

[2.1.1]

- Improvements
 - Add macro `FSL_FEATURE_DMAMUX_CHANNEL_NEEDS_ENDIAN_CONVERT` and `DMAMUX_CHANNEL_ENDIAN_CONVERTn` to do channel endian convert.

[2.1.0]

- Improvements
 - Modify the type of parameter source from `uint32_t` to `int32_t` in the `DMA_MUX_SetSource`.

[2.0.5]

- Improvements
 - Added feature `FSL_FEATURE_DMAMUX_CHCFG_REGISTER_WIDTH` for the difference of CHCFG register width.

[2.0.4]

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

[2.0.3]

- Bug Fixes
 - Fixed the issue for MISRA-2012 check.
 - * Fixed rule 10.4 and rule 10.3.

[2.0.2]

- New Features
 - Added an always-on enable feature to a DMA channel for ULP1 DMAMUX support.

[2.0.1]

- Bug Fixes
 - Fixed the build warning issue by changing the type of parameter source from `uint8_t` to `uint32_t` when setting DMA request source in `DMAMUX_SetSourceChange`.

[2.0.0]

- Initial version.

EWM

[2.0.4]

- Bug Fixes
 - Fixed CERT INT31-C violations.

[2.0.3]

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

[2.0.2]

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

[2.0.1]

- Bug Fixes
 - Fixed the hard fault in EWM_Deinit.

[2.0.0]

- Initial version.
-

FLASH

[3.3.0]

- New Feature
 - Support for EEPROM Quick Write on devices with FTFC

[3.2.0]

- New Feature
 - Basic support for FTFC

[3.1.3]

- New Feature
 - Support 512KB flash for Kinetis E serials.

[3.1.2]

- Bug Fixes — Remove redundant comments.

[3.1.1]

- Bug Fixes — MISRA C-2012 issue fixed: rule 10.3

[3.1.0]

- New Feature
 - Support erase flash asynchronously.

[3.0.2]

- Bug Fixes — MISRA C-2012 issue fixed: rule 8.4, 17.7, 10.4, 16.1, 21.15, 11.3, 10.7 — building warning -Wnull-dereference on arm compiler v6

[3.0.1]

- New Features
 - Added support FlexNVM alias for (kw37/38/39).

[3.0.0]

- Improvements
 - Reorganized FTFx flash driver source file.
 - Extracted flash cache driver from FTFx driver.
 - Extracted flexnvm flash driver from FTFx driver.

[2.3.1]

- Bug Fixes
 - Unified Flash IFR design from K3.
 - New encoding rule for K3 flash size.

[2.3.0]

- New Features
 - Added support for device with LP flash (K3S/G).
 - Added flash prefetch speculation APIs.
- Improvements
 - Refined flash_cache_clear function.
 - Reorganized the member of flash_config_t struct.

[2.2.0]

- New Features
 - Supported FTFL device in FLASH_Swap API.
 - Supported various pflash start addresses.
 - Added support for KV58 in cache clear function.

- Added support for device with secondary flash (KW40).
- Bug Fixes
 - Compiled execute-in-ram functions as PIC binary code for driver use.
 - Added missed flexram properties.
 - Fixed unaligned variable issue for execute-in-ram function code array.

[2.1.0]

- Improvements
 - Updated coding style to align with KSDK 2.0.
 - Different-alignment-size support for pflash and flexnvm.
 - Improved the implementation of execute-in-ram functions.

[2.0.0]

- Initial version
-

GPIO

[2.8.3]

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

[2.8.2]

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

[2.8.1]

- Bug Fixes
 - Fixed CERT INT31-C issues.

[2.8.0]

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

[2.8.0]

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

[2.7.3]

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

[2.7.2]

- New Features
 - Support devices without PORT module.

[2.7.1]

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

[2.7.0]

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

[2.6.0]

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

[2.5.3]

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

[2.5.2]

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

[2.5.1]

- Bug Fixes
 - Fixed wrong macro definition.
 - Fixed MISRA C-2012 rule issues in the FGPIO_CheckAttributeBytes() function.
 - Defined the new macro to separate the scene when the width of registers is different.
 - Removed some redundant macros.
- New Features

- Added some APIs to get/clear the interrupt status flag when the port doesn't control pins' interrupt.

[2.4.1]

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

[2.4.0]

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

[2.3.2]

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

[2.3.1]

- Improvements
 - Removed deprecated APIs.

[2.3.0]

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

[2.2.1]

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

[2.1.1]

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

[2.1.0]

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

I2C

[2.0.10]

- Bug Fixes
 - Fixed coverity issues.

[2.0.9]

- Bug Fixes
 - Fixed the MISRA-2012 violations.
 - * Fixed rule 8.4, 10.1, 10.4, 13.5, 20.8.

[2.0.8]

- Bug Fixes
 - Fixed the bug that DFEN bit of I2C Status register 2 could not be set in I2C_MasterInit.
 - MISRA C-2012 issue fixed: rule 14.2, 15.7, and 16.4.
 - Eliminated IAR Pa082 warnings from I2C_MasterTransferDMA and I2C_MasterTransferCallbackDMA by assigning volatile variables to local variables and using local variables instead.
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.3, 10.4, 11.9, 14.4, 15.7, 17.7.
- Improvements
 - Improved timeout mechanism when waiting certain state in transfer API.
 - Updated the I2C_WAIT_TIMEOUT macro to unified name I2C_RETRY_TIMES.
 - Moved the master manually acknowledge byte operation into static function I2C_MasterAckByte.
 - Fixed control/status clean flow issue inside I2C_MasterReadBlocking to avoid potential issue that pending status is cleaned before it's proceeded.

[2.0.7]

- Bug Fixes
 - Fixed the issue for MISRA-2012 check.
 - * Fixed rule 11.9 ,15.7 ,14.4 ,10.4 ,10.8 ,10.3, 10.1, 10.6, 13.5, 11.3, 13.2, 17.7, 5.7, 8.3, 8.5, 11.1, 16.1.
 - Fixed Coverity issue of unchecked return value in I2C_RTOS_Transfer.

- Fixed variable redefine issue by moving i2cBases from fsl_i2c.h to fsl_i2c.c.
- Improvements
 - Added I2C_MASTER_FACK_CONTROL macro to enable FACK control for master transfer receive flow with IP supporting double buffer, then master could hold the SCL by manually setting TX AK/NAK during data transfer.

[2.0.6]

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

[2.0.5]

- Improvements
 - Added I2C_WATI_TIMEOUT macro to allow the user to specify the timeout times for waiting flags in functional API and blocking transfer API.

[2.0.4]

- Bug Fixes
 - Added a proper handle for transfer config flag kI2C_TransferNoStartFlag to support transmit with kI2C_TransferNoStartFlag flag. Support write only or write+read with no start flag; does not support read only with no start flag.

[2.0.3]

- Bug Fixes
 - Removed enableHighDrive member in the master/slave configuration structure because the operation to HDRS bit is useless, the user need to use DSE bit in port register to configure the high drive capability.
 - Added register reset operation in I2C_MasterInit and I2C_SlaveInit APIs. Fixed issue where I2C could not switch between master and slave mode.
 - Improved slave IRQ handler to handle the corner case that stop flag and address match flag come synchronously.

[2.0.2]

- Bug Fixes
 - Fixed issue in master receive and slave transmit mode with no stop flag. The master could not succeed to start next transfer because the master could not send out re-start signal.
 - Fixed the out-of-order issue of data transfer due to memory barrier.
 - Added hold time configuration for slave. By leaving the SCL divider and MULT reset values when configured to slave mode, the setup and hold time of the slave is then reduced outside of spec for lower baudrates. This can cause intermittent arbitration loss on the master side.
- New Features

- Added address nak event for master.
- Added general call event for slave.

[2.0.1]

- New Features
 - Added double buffer enable configuration for SoCs which have the DFEN bit in S2 register.
 - Added flexible transmit/receive buffer size support in I2C_SlaveHandleIRQ.
 - Added start flag clear, address match, and release bus operation in I2C_SlaveWrite/ReadBlocking API.
- Bug Fixes
 - Changed the kI2C_SlaveRepeatedStartEvent to kI2C_SlaveStartEvent.

[2.0.0]

- Initial version.
-

IRTC

[2.3.3]

- Bug Fixes
 - Fix CERT INT31-C issue.

[2.3.2]

- Bug Fixes
 - Fixed API IRTC_GetDatetime read YEARMON, DAYS, HOURMIN, SECONDS registers issue.

[2.3.1]

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

[2.3.0]

- New Feature
 - Supported platforms with multiple IRTC instances.

[2.2.4]

- Bug Fixes
 - Fixed MISRA C-2012 issue 10.1, 10.3, 10.4, 10.7, 12.2.

[2.2.3]

- Bug Fixes
 - Updated undefined macro names by available ones.

[2.2.2]

- Bug Fixes
 - Fixed MISRA C-2012 issue 10.3.

[2.2.1]

- Bug Fixes
 - Fixed MISRA issues.

[2.2.0]

- New Feature
 - Add new APIs for CLK_SEL and CLKO to select RTC clock and enable/disable output to peripherals.
 - Supported platforms without tamper feature.

[2.1.3]

- Bug Fixes
 - Fixed MISRA C-2012 issue 10.1 and 10.4.

[2.1.2]

- Bug Fixes
 - Fixed kIRTC_TamperFlag flag can't be cleared issue.

[2.1.1]

- Bug Fixes
 - MISRA C-2012 issue check.
 - * Fixed rules, containing: rule-10.1, rule-10.3, rule-10.4.

[2.1.0]

- Bug Fixes
 - Fixed incorrect leap year check in IRTC_CheckDatetimeFormat.
- New Feature
 - Added new APIs for new feature FSL_FEATURE_RTC_HAS_SUBSYSTEM.
 - Added new APIs for TAMPER, TAMPER_QUEUE status get and clear.
 - Added new API to enable/disable 32 kHz RTC OSC clock during RTC register write.
 - Updated IRTC_SetTamperParams to support new feature FSL_FEATURE_RTC_HAS_FILTER23_CFG

- Updated `irtc_config_t` to exclude member `wakeupSelect` for new feature `FSL_FEATURE_RTC_HAS_NO_CTRL2_WAKEUP_MODE`.

[2.0.2]

- Bug Fixes
 - MISRA C-2012 issue check.
 - * Fixed rules, containing: rule-10.1, rule-10.3, rule-10.4, rule-10.6, rule-10.8, rule-11.9, rule-12.2, rule-15.5, rule-16.4, rule-17.7.

[2.0.1]

- Bug Fixes
 - Fixed the issue of hard code in `IRTC_Init`.

[2.0.0]

- Initial version.
-

LLWU

[2.0.5]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.3.
 - Fixed the issue that function `LLWU_SetExternalWakeupPinMode()` does not work on 32-bit width platforms.

[2.0.4]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.3, 10.4, 10.6, 10.7, 11.3.
 - Fixed issue that `LLWU_ClearExternalWakeupPinFlag` may clear other filter flags by mistake on platforms with 32-bit LLWU registers.

[2.0.3]

- Bug Fixes
 - Fixed MISRA-2012 rules.
 - * Rule 16.4.

[2.0.2]

- Improvements
 - Corrected driver function `LLWU_SetResetPinMode` parameter name.
- Bug Fixes
 - Fixed MISRA-2012 rules.
 - * Rule 14.4, 10.8, 10.4, 10.3.

[2.0.1]

- Other Changes
 - Updates for KL8x.

[2.0.0]

- Initial version.
-

LPTMR**[2.2.1]**

- Bug Fixes
 - Fix CERT INT31-C issues.

[2.2.0]

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

[2.1.1]

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

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

LPUART

[2.10.0]

- New Feature
 - Added support to configure RTS watermark.

[2.9.4]

- Improvements
 - Merged duplicate code.

[2.9.3]

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

[2.9.2]

- Bug Fixes
 - Fixed coverity issues.

[2.9.1]

- Bug Fixes
 - Fixed coverity issues.

[2.9.0]

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

[2.8.3]

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

[2.8.2]

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

[2.8.1]

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

[2.8.0]

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

[2.7.7]

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

[2.7.6]

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

[2.7.5]

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

[2.7.4]

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

[2.7.3]

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

[2.7.2]

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

[2.7.1]

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

[2.7.0]

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

[2.6.0]

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

[2.5.3]

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

[2.5.2]

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

[2.5.1]

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

[2.5.0]

- Bug Fixes
 - Added missing interrupt enable masks kLPUART_Match1InterruptEnable and kLPUART_Match2InterruptEnable.
 - Fixed bug in LPUART_EnableInterrupts, LPUART_DisableInterrupts and LPUART_GetEnabledInterrupts that the BAUD[LBKDIE] bit field should be soc specific.
 - Fixed bug in LPUART_TransferHandleIRQ that idle line interrupt should be disabled when rx data size is zero.

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

[2.4.1]

- Bug Fixes
 - Fixed MISRA advisory 17.7 issues.

[2.4.0]

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

[2.3.1]

- Bug Fixes
 - Fixed MISRA advisory 15.5 issues.

[2.3.0]

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

[2.2.8]

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

[2.2.7]

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

[2.2.6]

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

[2.2.5]

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

[2.2.4]

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

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.1]

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

[2.1.0]

- Improvements
 - Update transactional APIs.
-

LPUART_DMA

[2.4.0]

- Refer LPUART driver change log 2.1.0 to 2.4.0
-

MCM

[2.2.0]

- Improvements
 - Support platforms with less features.

[2.1.0]

- Others
 - Remove byteID from mcm_lmem_fault_attribute_t for document update.

[2.0.0]

- Initial version.
-

MMAU

[2.0.1]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 7.3, 10.1, 10.3, 10.4.

[2.0.0]

- Initial version.
-

PDB

[2.0.4]

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

[2.0.3]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 17.7.

[2.0.2]

- Improvement:
 - Used macros in feature file instead of that in IO map.

[2.0.1]

- Changed PDB register base array to const.

[2.0.0]

- Initial version.
-

PIT**[2.2.0]**

- Bug Fixes
 - According to ERR050763, PIT_LDVAL_STAT register is not reliable in dynamic load mode, so remove the status check in PIT_SetRtiTimerPeriod which added since 2.1.1.
 - Removed not used bit PIT_RTI_TCTRL_CHN_MASK.
- Improvements
 - Added more guide about get RTI load status in PIT_SetRtiTimerPeriod's API comment.
 - Change PIT_RTI_Deinit to inline API.
 - Ensure PIT peripheral clock enabled in PIT_RTI_Init.
- New Features
 - Added PIT_ClearRtiSyncStatus API to clear the RTI_LDVAL_STAT register.

[2.1.1]

- Bug Fixes
 - Enable PIT when using RTI to ensure RTI can work properly in debug mode.
- Improvements
 - Added status check in PIT_SetRtiTimerPeriod to ensure the load value is synchronized into the RTI clock domain.
 - Added note for PIT_RTI_Init to remind users wait RTI sync.

[2.1.0]

- New Features
 - Support RTI (Real Time Interrupt) timer.

[2.0.5]

- Improvements
 - Support workaround for ERR007914. This workaround guarantee the write to MCR register is not ignored.

[2.0.4]

- Bug Fixes
 - Fixed PIT_SetTimerPeriod implementation, the load value trigger should be PIT clock cycles minus 1.

[2.0.3]

- Bug Fixes
 - Clear all status bits for all channels to make sure the status of all TCTRL registers is clean.

[2.0.2]

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

[2.0.1]

- Bug Fixes
 - Cleared timer enable bit for all channels in function PIT_Init() to make sure all channels stay in disable status before setting other configurations.
 - Fixed MISRA-2012 rules.
 - * Rule 14.4, rule 10.4.

[2.0.0]

- Initial version.
-

PMC

[2.0.3]

- Bug Fixes
 - Fixed the violation of MISRA C-2012 rule 11.3.

[2.0.2]

- Bug Fixes
 - Fixed the violations of MISRA 2012 rules:
 - * Rule 10.3.

[2.0.1]

- Bug Fixes
 - Fixed MISRA issues.
 - * Rule 10.8, Rule 10.3.

[2.0.0]

- Initial version.
-

PORT

[2.5.1]

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

[2.5.0]

- Bug Fixes
 - Correct the kPORT_MuxAsGpio for some platforms.

[2.4.1]

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

[2.4.0]

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

[2.3.0]

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

[2.2.0]

- New Features
 - Added new api PORT_EnablePinDoubleDriveStrength.

[2.1.1]

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

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

QTMR

[2.0.2]

- Bug Fixes
 - Fix CERT INT30-C and CERT INT31-C violations.

[2.0.1]

- Bug Fixes
 - MISRA C-2012 issue check.
 - * Fixed rules, containing: rule-10.1, rule-10.3, rule-10.4, rule-11.9, rule-14.4, rule-15.5, rule-17.7.
 - Changed FSL_COMPONENT_ID as platform.drivers.qtmr_2.

[2.0.0]

- Initial version.
-

RCM

[2.0.4]

- Bug Fixes
 - Fixed violation of MISRA C-2012 rule 10.3

[2.0.3]

- Bug Fixes
 - Fixed violation of MISRA C-2012 rules.

[2.0.2]

- Bug Fixes
 - Fixed MISRA issue.
 - * Rule 10.8, rule 10.1, rule 13.2, rule 3.1.

[2.0.1]

- Bug Fixes
 - Fixed kRCM_SourceSw bit shift issue.

[2.0.0]

- Initial version.
-

RNGA**[2.0.2]**

- Bug fix:
 - Fix MISRA issue.

[2.0.1]

- Bug fix:
 - Fixed C++ build warning in RNGA driver.

[2.0.0]

- Initial version.
-

SIM**[2.2.0]**

- Improvements
 - Added API to trigger TRGMUX.

[2.1.3]

- Improvements
 - Updated function SIM_GetUniqueId to support different register names.

[2.1.2]

- Bug Fixes
 - Fixed SIM_GetUniqueId bug that could not get UIDH.

[2.1.1]

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

[2.1.0]

- Improvements
 - Added new APIs: SIM_GetRfAddr() and SIM_EnableSystickClock().

[2.0.0]

- Initial version.
-

SLCD

[2.1.0]

- New Features
 - Added new enumerations, updated SLCD_Init and SLCD_GetDefaultConfig to support new low power IP on new SoCs.

[2.0.4]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.4.

[2.0.3]

- Bug Fixes
 - Fixed SLCD_Init bug that some bit-fields are cleared by mistake.

[2.0.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 3.1, 10.1, 10.3, 10.3, 10.4 11.4, 17.7

[2.0.1]

- Bug Fixes
 - Changed the Blink mode start setting flow.
- Other Changes
 - Added static to SLCD global variables.

[2.0.0]

- Initial version.
-

SMC

[2.0.7]

- Bug Fixes
 - Fixed MISRA-2012 issue 10.3.

[2.0.6]

- Bug Fixes
 - Fixed issue for MISRA-2012 check.
 - * Fixed rule 10.3, rule 11.3.

[2.0.5]

- Bug Fixes
 - Fixed issue for MISRA-2012 check.
 - * Fixed rule 15.7, rule 14.4, rule 10.3, rule 10.1, rule 10.4.

[2.0.4]

- Bug Fixes
 - When entering stop modes, used RAM function for the flash synchronization issue. Application should make sure that, the RW data of fsl_smc.c is located in memory region which is not powered off in stop modes.

[2.0.3]

- Improvements
 - Added APIs SMC_PreEnterStopModes, SMC_PreEnterWaitModes, SMC_PostExitWaitModes, and SMC_PostExitStopModes.

[2.0.2]

- Bug Fixes
 - Added DSB before WFI while ISB after WFI.
- Other Changes
 - Updated SMC_SetPowerModeVlpw implementation.

[2.0.1]

- Other Changes
 - Updated for KL8x.

[2.0.0]

- Initial version.
-

SPI**[2.1.4]**

- Bug Fixes
 - Fixed coverity issues.

[2.1.3]

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

[2.1.2]

- Improvements
 - Changed SPI_DUMMYDATA to 0x00.

[2.1.1]

- Bug Fixes
 - Fixed MISRA 10.3 violation.

[2.1.0]

- Improvements
 - Added timeout mechanism when waiting certain states in transfer driver.
- Bug Fixes
 - Fixed the bug that, when working as a slave, instance that does not have FIFO may miss some rx data.
 - Fixed master RX data overflow issue by synchronizing transmit and receive process.
 - Fixed issue that slave should not share the same non-blocking initialization API and IRQ handler with master to prevent dead lock issue.
 - Fixed issue that callback should be invoked after all data is sent out to bus.
 - Added code in SPI_SlaveTransferNonBlocking to empty rx buffer before initializing transfer.

[2.0.5]

- Bug Fixes
 - Eliminated Pa082 warnings from SPI_WriteNonBlocking and SPI_GetStatusFlags.
 - Fixed MISRA issues.
 - * Fixed issues 10.1, 10.3, 10.4, 10.7, 10.8, 11.9, 14.4, 17.7.

[2.0.4]

- New Features
 - Supported 3-wire mode for SPI driver. Added new API SPI_SetPinMode() to control the transfer direction of the single wire. For master instance, MOSI is selected as I/O pin. For slave instance, MISO is selected as I/O pin.
 - Added dummy data setup API to allow users to configure the dummy data to be transferred.

[2.0.3]

- Bug Fixes
 - Fixed the potential interrupt race condition at high baudrate when calling API `SPI_MasterTransferNonBlocking`.

[2.0.2]

- New Features
 - Allowed users to set the transfer size for `SPI_TransferNoBlocking` non-integer times of watermark.
 - Allowed users to define the dummy data. Users only need to define the macro `SPI_DUMMYDATA` in applications.

[2.0.1]

- Bug Fixes
 - Fixed `SPI_Enable` function parameter error.
 - Set the `s_dummy` variable as static variable in `fsl_spi_dma.c`.
- Improvements
 - Optimized the code size while not using transactional API.
 - Improved performance in polling method.
 - Added `#ifndef/#endif` to allow users to change the default tx value at compile time.

[2.0.0]

- Initial version.
-

SPI DMA Driver**[2.1.1]**

- Bug Fixes
 - Fixed the bug that TX data not sent to bus when transfer finish callback is called.

[2.1.0]

- Improvements
 - Added timeout mechanism when waiting certain states in transfer driver.
- Bug Fixes
 - Fixed the bug that, when working as a slave, instance that does not have FIFO may miss some rx data.
 - Fixed master RX data overflow issue by synchronizing transmit and receive process.
 - Fixed issue that slave should not share the same non-blocking initialization API and IRQ handler with master to prevent dead lock issue.
 - Fixed issue that callback should be invoked after all data is sent out to bus.

- Added code in SPI_SlaveTransferNonBlocking to empty rx buffer before initializing transfer.

[2.0.5]

- Bug Fixes
 - Eliminated Pa082 warnings from SPI_WriteNonBlocking and SPI_GetStatusFlags.
 - Fixed MISRA issues.
 - * Fixed issues 10.1, 10.3, 10.4, 10.7, 10.8, 11.9, 14.4, 17.7.

[2.0.4]

- New Features
 - Supported 3-wire mode for SPI driver. Added new API SPI_SetPinMode() to control the transfer direction of the single wire. For master instance, MOSI is selected as I/O pin. For slave instance, MISO is selected as I/O pin.
 - Added dummy data setup API to allow users to configure the dummy data to be transferred.

[2.0.3]

- Bug Fixes
 - Fixed the potential interrupt race condition at high baudrate when calling API SPI_MasterTransferNonBlocking.

[2.0.2]

- New Features
 - Allowed users to set the transfer size for SPI_TransferNoBlocking non-integer times of watermark.
 - Allowed users to define the dummy data. Users only need to define the macro SPI_DUMMYDATA in applications.

[2.0.1]

- Bug Fixes
 - Fixed SPI_Enable function parameter error.
 - Set the s_dummy variable as static variable in fsl_spi_dma.c.
- Improvements
 - Optimized the code size while not using transactional API.
 - Improved performance in polling method.
 - Added #ifndef/#endif to allow users to change the default tx value at compile time.

[2.0.0]

- Initial version.

SYSMPU

[2.2.3]

- Bug Fixes
 - Fixed violation of MISRA C-2012 Rule 10.4, a part of issues is ignored before.

[2.2.2]

- Bug Fixes
 - Fixed violation of MISRA C-2012 Rule 10.1, 10.3, 10.4, 10.7, 10.6, 10.8, 12.2.

[2.2.1]

- Bug Fixes
 - Fixed MISRA issue.

[2.2.0]

- Improvements
 - Renamed MPU to SYSMPU.
 - Changed macro definition for slave number and fixed the get error status calculation.

[2.1.1]

- Improvements
 - Added the feature file macro definition limitation for the MPU_SetRegionRwMasterAccessRights().

[2.1.0]

- Other Changes
 - API changes:
 - * Changed the mpu_region_num_t and mpu_master_t to uint32_t.
 - * Changed the mpu_low_masters_access_rights_t, mpu_high_masters_access_rights_t to mpu_rwxrights_master_access_control_t, mpu_rwrights_master_access_control_t.
 - * Changed the MPU_SetRegionLowMasterAccessRights(), MPU_SetRegionHighMasterAccessRights() to MPU_SetRegionRwxMasterAccessRights(), MPU_SetRegionRwMasterAccessRights().

[2.0.0]

- Initial version.
-

UART

[2.5.1]

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

[2.5.0]

- New Features
 - Added APIs `UART_GetRxFifoCount`/`UART_GetTxFifoCount` to get rx/tx FIFO data count.
 - Added APIs `UART_SetRxFifoWatermark`/`UART_SetTxFifoWatermark` to set rx/tx FIFO water mark.
- Bug Fixes
 - Fixed bug of race condition during UART transfer using transactional APIs, by disabling and re-enabling the global interrupt before and after critical operations on interrupt enable registers.
 - Fixed DMA/eDMA transfer blocking issue by enabling tx idle interrupt after DMA/eDMA transmission finishes.

[2.4.0]

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

[2.3.0]

- Bug Fixes
 - Fixed the bug that, when framing/parity/noise/overflow flag or idle line detect flag is set, receive FIFO should be flushed to avoid FIFO pointer being in unknown state, since FIFO has no valid data.
- Improvements
 - Modified `UART_TransferHandleIRQ` so that `txState` will be set to idle only when all data has been sent out to bus.
 - Modified `UART_TransferGetSendCount` so that this API returns the real byte count that UART has sent out rather than the software buffer status.
 - Added timeout mechanism when waiting for certain states in transfer driver.

[2.2.0]

- New Features
 - Added UART hardware FIFO enable/disable API.
- Improvements

- Added check for `kUART_TransmissionCompleteFlag` in `UART_TransferHandleIRQ`, `UART_SendEDMACallback` and `UART_TransferSendDMACallback` to ensure all the data would be sent out to bus.
- Bug Fixes
 - Eliminated IAR Pa082 warnings from `UART_TransferGetRxRingBufferLength`, `UART_GetEnabledInterrupts`, `UART_GetStatusFlags` and `UART_TransferHandleIRQ`.
 - Added code in `UART_ReadBlocking` so that if more than one receiver errors occur, all status flags will be cleared and the most severe error status will be returned.
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.3, 10.4, 14.4, 11.6, 17.7.

[2.1.6]

- Bug Fixes
 - Fixed the issue of register's being in repeatedly reading status while performing the IRQ routine.

[2.1.5]

- Improvements
 - Added hardware flow control function support.
 - Added idle-line-detecting feature in `UART_TransferNonBlocking` function. If an idle line is detected, a callback will be triggered with status `kStatus_UART_IdleLineDetected` returned. This feature may be useful when the number of received bytes is less than the expected receive data size. Before triggering the callback, data in the FIFO is read out (if it has FIFO), and no interrupt will be disabled except for the case that the receive data size reaches 0.
 - Enabled the RX FIFO watermark function. With the idle-line-detecting feature enabled, you can set the watermark value to whatever you want (should not be bigger than the RX FIFO size). Data is then received and a callback will be triggered when data receive ends.

[2.1.4]

- Improvements
 - Changed parameter type in `UART_RTOS_Init()` struct `rtos_uart_config` → `uart_rtos_config_t`.
- Bug Fixes
 - Disabled UART receive interrupt instead of global interrupt when reading data from ring buffer. With ring buffer used, receive nonblocking will disable global interrupt to protect the ring buffer. This has a negative effect on other IPs using interrupt.

[2.1.3]

- New Features
 - Added RX framing error and parity error status check when using interrupt transfer.

[2.1.2]

- Bug Fixes
 - Fixed baud rate fine adjust bug to make the computed baud rate more accurate.

[2.1.1]

- Bug Fixes
 - Removed needless check of event flags and assert in UART_RTOS_Receive.
 - Always waited for RX event flag in UART_RTOS_Receive.

[2.1.0]

- Improvements
 - Added transactional API.

[2.0.0]

- Initial version.
-

UART_DMA

[2.5.0]

- Refer UART driver change log 2.1.0 to 2.5.0
-

VREF

[2.1.3]

- Improvements
 - Add timeout for APIs with dfmea issues.

[2.1.2]

- Bug Fixes
 - Fixed the violation of MISRA-2012 rule 10.3.
 - Fixed MISRA C-2012 rule 10.3, rule 10.4 violation.

[2.1.1]

- Bug Fixes
 - MISRA-2012 issue fixed.
 - * Fixed rules containing: rule-10.4, rule-10.3, rule-10.1.

[2.1.0]

- Improvements
 - Added new functions to support L5K board: added VREF_SetTrim2V1Val() and VREF_GetTrim2V1Val() functions to supply 2V1 output mode.

[2.0.0]

- Initial version.
-

WDOG**[2.0.2]**

- Improvements
 - WDG_Init() adds a 256 bus clock delay for WCT window finish.

[2.0.1]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.3, 10.4, 10.6, 10.7 11.9 and 17.7.

[2.0.0]

- Initial version.
-

XBAR**[2.1.0]**

- Improvements
 - Improved to support XBAR which has less than 4 interrupt output.
- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 12.2.

[2.0.5]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.1, 10.3, 10.4, 10.6, 10.7, 10.8, 12.2, 18.1, 20.7.

[2.0.4]

- Bug Fixes
 - Fixed IAR build warning Pa082.

[2.0.3]

- Improvements
 - Optimized XBAR_SetOutputSignalConfig.

[2.0.2]

- Bug Fixes
 - Corrected configuration for function XBAR_SetOutputSignalConfig.

[2.0.1]

- Bug Fixes
 - Fixed w1c bits for XBAR_SetOutputSignalConfig function.

[2.0.0]

- Initial version.
-

1.6 Driver API Reference Manual

This section provides a link to the Driver API RM, detailing available drivers and their usage to help you integrate hardware efficiently.

[MKM35Z7](#)

1.7 Middleware Documentation

Find links to detailed middleware documentation for key components. While not all onboard middleware is covered, this serves as a useful reference for configuration and development.

1.7.1 FreeMASTER

[freemaster](#)

1.7.2 FreeRTOS

[FreeRTOS](#)

Chapter 2

MKM35Z7

2.1 ADC16: 16-bit SAR Analog-to-Digital Converter Driver

`void ADC16_Init(ADC_Type *base, const adc16_config_t *config)`

Initializes the ADC16 module.

Parameters

- `base` – ADC16 peripheral base address.
- `config` – Pointer to configuration structure. See “*adc16_config_t*”.

`void ADC16_Deinit(ADC_Type *base)`

De-initializes the ADC16 module.

Parameters

- `base` – ADC16 peripheral base address.

`void ADC16_GetDefaultConfig(adc16_config_t *config)`

Gets an available pre-defined settings for the converter’s configuration.

This function initializes the converter configuration structure with available settings. The default values are as follows.

```
config->referenceVoltageSource = kADC16_ReferenceVoltageSourceVref;
config->clockSource           = kADC16_ClockSourceAsynchronousClock;
config->enableAsynchronousClock = false;
config->clockDivider          = kADC16_ClockDivider8;
config->resolution             = kADC16_ResolutionSE12Bit;
config->longSampleMode         = kADC16_LongSampleDisabled;
config->enableHighSpeed        = false;
config->enableLowPower         = false;
config->enableContinuousConversion = false;
```

Parameters

- `config` – Pointer to the configuration structure.

`status_t ADC16_DoAutoCalibration(ADC_Type *base)`

Automates the hardware calibration.

This auto calibration helps to adjust the plus/minus side gain automatically. Execute the calibration before using the converter. Note that the hardware trigger should be used during the calibration.

Parameters

- `base` – ADC16 peripheral base address.

Return values

- `kStatus_Success` – Calibration is done successfully.
- `kStatus_Fail` – Calibration has failed.

Returns

Execution status.

`static inline void ADC16_SetOffsetValue(ADC_Type *base, int16_t value)`

Sets the offset value for the conversion result.

This offset value takes effect on the conversion result. If the offset value is not zero, the reading result is subtracted by it. Note, the hardware calibration fills the offset value automatically.

Parameters

- `base` – ADC16 peripheral base address.
- `value` – Setting offset value.

`static inline void ADC16_EnableDMA(ADC_Type *base, bool enable)`

Enables generating the DMA trigger when the conversion is complete.

Parameters

- `base` – ADC16 peripheral base address.
- `enable` – Switcher of the DMA feature. “true” means enabled, “false” means not enabled.

`static inline void ADC16_EnableHardwareTrigger(ADC_Type *base, bool enable)`

Enables the hardware trigger mode.

Parameters

- `base` – ADC16 peripheral base address.
- `enable` – Switcher of the hardware trigger feature. “true” means enabled, “false” means not enabled.

`void ADC16_SetChannelMuxMode(ADC_Type *base, adc16_channel_mux_mode_t mode)`

Sets the channel mux mode.

Some sample pins share the same channel index. The channel mux mode decides which pin is used for an indicated channel.

Parameters

- `base` – ADC16 peripheral base address.
- `mode` – Setting channel mux mode. See “`adc16_channel_mux_mode_t`”.

`void ADC16_SetHardwareCompareConfig(ADC_Type *base, const
adc16_hardware_compare_config_t *config)`

Configures the hardware compare mode.

The hardware compare mode provides a way to process the conversion result automatically by using hardware. Only the result in the compare range is available. To compare the range, see “`adc16_hardware_compare_mode_t`” or the appropriate reference manual for more information.

Parameters

- `base` – ADC16 peripheral base address.

- `config` – Pointer to the “`adc16_hardware_compare_config_t`” structure. Passing “NULL” disables the feature.

`void ADC16_SetHardwareAverage(ADC_Type *base, adc16_hardware_average_mode_t mode)`

Sets the hardware average mode.

The hardware average mode provides a way to process the conversion result automatically by using hardware. The multiple conversion results are accumulated and averaged internally making them easier to read.

Parameters

- `base` – ADC16 peripheral base address.
- `mode` – Setting the hardware average mode. See “`adc16_hardware_average_mode_t`”.

`void ADC16_SetPGAConfig(ADC_Type *base, const adc16_pga_config_t *config)`

Configures the PGA for the converter’s front end.

Parameters

- `base` – ADC16 peripheral base address.
- `config` – Pointer to the “`adc16_pga_config_t`” structure. Passing “NULL” disables the feature.

`uint32_t ADC16_GetStatusFlags(ADC_Type *base)`

Gets the status flags of the converter.

Parameters

- `base` – ADC16 peripheral base address.

Returns

Flags’ mask if indicated flags are asserted. See “`_adc16_status_flags`”.

`void ADC16_ClearStatusFlags(ADC_Type *base, uint32_t mask)`

Clears the status flags of the converter.

Parameters

- `base` – ADC16 peripheral base address.
- `mask` – Mask value for the cleared flags. See “`_adc16_status_flags`”.

`static inline void ADC16_EnableAsynchronousClockOutput(ADC_Type *base, bool enable)`

Enable/disable ADC Asynchronous clock output to other modules.

Parameters

- `base` – ADC16 peripheral base address.
- `enable` – Used to enable/disable ADC ADACK output.
 - **true** Asynchronous clock and clock output is enabled regardless of the state of the ADC.
 - **false** Asynchronous clock output disabled, asynchronous clock is enabled only if it is selected as input clock and a conversion is active.

`void ADC16_SetChannelConfig(ADC_Type *base, uint32_t channelGroup, const adc16_channel_config_t *config)`

Configures the conversion channel.

This operation triggers the conversion when in software trigger mode. When in hardware trigger mode, this API configures the channel while the external trigger source helps to trigger the conversion.

Note that the “Channel Group” has a detailed description. To allow sequential conversions of the ADC to be triggered by internal peripherals, the ADC has more than one group of status and control registers, one for each conversion. The channel group parameter indicates which group of registers are used, for example, channel group 0 is for Group A registers and channel group 1 is for Group B registers. The channel groups are used in a “ping-pong” approach to control the ADC operation. At any point, only one of the channel groups is actively controlling ADC conversions. The channel group 0 is used for both software and hardware trigger modes. Channel group 1 and greater indicates multiple channel group registers for use only in hardware trigger mode. See the chip configuration information in the appropriate MCU reference manual for the number of SC1n registers (channel groups) specific to this device. Channel group 1 or greater are not used for software trigger operation. Therefore, writing to these channel groups does not initiate a new conversion. Updating the channel group 0 while a different channel group is actively controlling a conversion is allowed and vice versa. Writing any of the channel group registers while that specific channel group is actively controlling a conversion aborts the current conversion.

Parameters

- base – ADC16 peripheral base address.
- channelGroup – Channel group index.
- config – Pointer to the “adc16_channel_config_t” structure for the conversion channel.

```
static inline uint32_t ADC16_GetChannelConversionValue(ADC_Type *base, uint32_t
                                                    channelGroup)
```

Gets the conversion value.

Parameters

- base – ADC16 peripheral base address.
- channelGroup – Channel group index.

Returns

Conversion value.

```
uint32_t ADC16_GetChannelStatusFlags(ADC_Type *base, uint32_t channelGroup)
```

Gets the status flags of channel.

Parameters

- base – ADC16 peripheral base address.
- channelGroup – Channel group index.

Returns

Flags’ mask if indicated flags are asserted. See “_adc16_channel_status_flags”.

```
FSL_ADC16_DRIVER_VERSION
```

ADC16 driver version 2.3.0.

```
enum _adc16_channel_status_flags
```

Channel status flags.

Values:

```
enumerator kADC16_ChannelConversionDoneFlag
```

Conversion done.

```
enum _adc16_status_flags
```

Converter status flags.

Values:

enumerator kADC16_ActiveFlag

Converter is active.

enumerator kADC16_CalibrationFailedFlag

Calibration is failed.

enum _adc_channel_mux_mode

Channel multiplexer mode for each channel.

For some ADC16 channels, there are two pin selections in channel multiplexer. For example, ADC0_SE4a and ADC0_SE4b are the different channels that share the same channel number.

Values:

enumerator kADC16_ChannelMuxA

For channel with channel mux a.

enumerator kADC16_ChannelMuxB

For channel with channel mux b.

enum _adc16_clock_divider

Clock divider for the converter.

Values:

enumerator kADC16_ClockDivider1

For divider 1 from the input clock to the module.

enumerator kADC16_ClockDivider2

For divider 2 from the input clock to the module.

enumerator kADC16_ClockDivider4

For divider 4 from the input clock to the module.

enumerator kADC16_ClockDivider8

For divider 8 from the input clock to the module.

enum _adc16_resolution

Converter's resolution.

Values:

enumerator kADC16_Resolution8or9Bit

Single End 8-bit or Differential Sample 9-bit.

enumerator kADC16_Resolution12or13Bit

Single End 12-bit or Differential Sample 13-bit.

enumerator kADC16_Resolution10or11Bit

Single End 10-bit or Differential Sample 11-bit.

enumerator kADC16_ResolutionSE8Bit

Single End 8-bit.

enumerator kADC16_ResolutionSE12Bit

Single End 12-bit.

enumerator kADC16_ResolutionSE10Bit

Single End 10-bit.

enumerator kADC16_ResolutionDF9Bit

Differential Sample 9-bit.

enumerator kADC16__ResolutionDF13Bit
Differential Sample 13-bit.

enumerator kADC16__ResolutionDF11Bit
Differential Sample 11-bit.

enum __adc16__clock__source
Clock source.

Values:

enumerator kADC16__ClockSourceAlt0
Selection 0 of the clock source.

enumerator kADC16__ClockSourceAlt1
Selection 1 of the clock source.

enumerator kADC16__ClockSourceAlt2
Selection 2 of the clock source.

enumerator kADC16__ClockSourceAlt3
Selection 3 of the clock source.

enumerator kADC16__ClockSourceAsynchronousClock
Using internal asynchronous clock.

enum __adc16__long__sample__mode
Long sample mode.

Values:

enumerator kADC16__LongSampleCycle24
20 extra ADCK cycles, 24 ADCK cycles total.

enumerator kADC16__LongSampleCycle16
12 extra ADCK cycles, 16 ADCK cycles total.

enumerator kADC16__LongSampleCycle10
6 extra ADCK cycles, 10 ADCK cycles total.

enumerator kADC16__LongSampleCycle6
2 extra ADCK cycles, 6 ADCK cycles total.

enumerator kADC16__LongSampleDisabled
Disable the long sample feature.

enum __adc16__reference__voltage__source
Reference voltage source.

Values:

enumerator kADC16__ReferenceVoltageSourceVref
For external pins pair of VrefH and VrefL.

enumerator kADC16__ReferenceVoltageSourceValt
For alternate reference pair of ValtH and ValtL.

enumerator kADC16__ReferenceVoltageSourceBandgap
For bandgap voltage from PMC.

enum __adc16__hardware__average__mode
Hardware average mode.

Values:

enumerator kADC16__HardwareAverageCount4

For hardware average with 4 samples.

enumerator kADC16__HardwareAverageCount8

For hardware average with 8 samples.

enumerator kADC16__HardwareAverageCount16

For hardware average with 16 samples.

enumerator kADC16__HardwareAverageCount32

For hardware average with 32 samples.

enumerator kADC16__HardwareAverageDisabled

Disable the hardware average feature.

enum __adc16__hardware__compare__mode

Hardware compare mode.

Values:

enumerator kADC16__HardwareCompareMode0

$x < \text{value1}$.

enumerator kADC16__HardwareCompareMode1

$x > \text{value1}$.

enumerator kADC16__HardwareCompareMode2

if $\text{value1} \leq \text{value2}$, then $x < \text{value1} \mid \mid x > \text{value2}$; else, $\text{value1} > x > \text{value2}$.

enumerator kADC16__HardwareCompareMode3

if $\text{value1} \leq \text{value2}$, then $\text{value1} \leq x \leq \text{value2}$; else $x \geq \text{value1} \mid \mid x \leq \text{value2}$.

enum __adc16__pga__gain

PGA's Gain mode.

Values:

enumerator kADC16__PGAGainValueOf1

For amplifier gain of 1.

enumerator kADC16__PGAGainValueOf2

For amplifier gain of 2.

enumerator kADC16__PGAGainValueOf4

For amplifier gain of 4.

enumerator kADC16__PGAGainValueOf8

For amplifier gain of 8.

enumerator kADC16__PGAGainValueOf16

For amplifier gain of 16.

enumerator kADC16__PGAGainValueOf32

For amplifier gain of 32.

enumerator kADC16__PGAGainValueOf64

For amplifier gain of 64.

typedef enum __adc_channel_mux_mode adc16_channel_mux_mode_t

Channel multiplexer mode for each channel.

For some ADC16 channels, there are two pin selections in channel multiplexer. For example, ADC0_SE4a and ADC0_SE4b are the different channels that share the same channel number.

```
typedef enum _adc16_clock_divider adc16_clock_divider_t
    Clock divider for the converter.
typedef enum _adc16_resolution adc16_resolution_t
    Converter's resolution.
typedef enum _adc16_clock_source adc16_clock_source_t
    Clock source.
typedef enum _adc16_long_sample_mode adc16_long_sample_mode_t
    Long sample mode.
typedef enum _adc16_reference_voltage_source adc16_reference_voltage_source_t
    Reference voltage source.
typedef enum _adc16_hardware_average_mode adc16_hardware_average_mode_t
    Hardware average mode.
typedef enum _adc16_hardware_compare_mode adc16_hardware_compare_mode_t
    Hardware compare mode.
typedef enum _adc16_pga_gain adc16_pga_gain_t
    PGA's Gain mode.
typedef struct _adc16_config adc16_config_t
    ADC16 converter configuration.
typedef struct _adc16_hardware_compare_config adc16_hardware_compare_config_t
    ADC16 Hardware comparison configuration.
typedef struct _adc16_channel_config adc16_channel_config_t
    ADC16 channel conversion configuration.
typedef struct _adc16_pga_config adc16_pga_config_t
    ADC16 programmable gain amplifier configuration.
struct _adc16_config
    #include <fsl_adc16.h> ADC16 converter configuration.
```

Public Members

```
adc16_reference_voltage_source_t referenceVoltageSource
    Select the reference voltage source.
adc16_clock_source_t clockSource
    Select the input clock source to converter.
bool enableAsynchronousClock
    Enable the asynchronous clock output.
adc16_clock_divider_t clockDivider
    Select the divider of input clock source.
adc16_resolution_t resolution
    Select the sample resolution mode.
adc16_long_sample_mode_t longSampleMode
    Select the long sample mode.
bool enableHighSpeed
    Enable the high-speed mode.
```


bool enableLowPower
 Enable low power.

bool enableContinuousConversion
 Enable continuous conversion mode.

adc16_hardware_average_mode_t hardwareAverageMode
 Set hardware average mode.

struct __adc16_hardware_compare_config
 #include <fsl_adc16.h> ADC16 Hardware comparison configuration.

Public Members

adc16_hardware_compare_mode_t hardwareCompareMode
 Select the hardware compare mode. See “adc16_hardware_compare_mode_t”.

int16_t value1
 Setting value1 for hardware compare mode.

int16_t value2
 Setting value2 for hardware compare mode.

struct __adc16_channel_config
 #include <fsl_adc16.h> ADC16 channel conversion configuration.

Public Members

uint32_t channelNumber
 Setting the conversion channel number. The available range is 0-31. See channel connection information for each chip in Reference Manual document.

bool enableInterruptOnConversionCompleted
 Generate an interrupt request once the conversion is completed.

bool enableDifferentialConversion
 Using Differential sample mode.

struct __adc16_pga_config
 #include <fsl_adc16.h> ADC16 programmable gain amplifier configuration.

Public Members

adc16_pga_gain_t pgaGain
 Setting PGA gain.

bool enableRunInNormalMode
 Enable PGA working in normal mode, or low power mode by default.

bool disablePgaChopping
 Disable the PGA chopping function. The PGA employs chopping to remove/reduce offset and 1/f noise and offers an offset measurement configuration that aids the offset calibration.

bool enableRunInOffsetMeasurement
 Enable the PGA working in offset measurement mode. When this feature is enabled, the PGA disconnects itself from the external inputs and auto-configures into offset measurement mode. With this field set, run the ADC in the recommended settings and enable the maximum hardware averaging to get the PGA offset number. The output is the (PGA offset * (64+1)) for the given PGA setting.

2.2 AFE: Analog Front End Driver

`void AFE_Init(AFE_Type *base, const afe_config_t *config)`

Initialization for the AFE module.

This function configures the AFE module for the configuration which are shared by all channels.

Parameters

- `base` – AFE peripheral base address.
- `config` – Pointer to structure of “*afe_config_t*”.

`void AFE_Deinit(AFE_Type *base)`

De-Initialization for the AFE module.

This function disables clock.

Parameters

- `base` – AFE peripheral base address.

`void AFE_GetDefaultConfig(afe_config_t *config)`

Fills the user configure structure.

This function fills the *afe_config_t* structure with default settings. Default value are:

```
config->enableLowPower = false;
config->resultFormat    = kAFE_ResultFormatRight;
config->clockDivider    = kAFE_ClockDivider2;
config->clockSource     = kAFE_ClockSource1;
config->startupCount    = 2U;
```

Parameters

- `config` – Pointer to structure of “*afe_config_t*”.

`static inline void AFE_SoftwareReset(AFE_Type *base, bool enable)`

Software reset the AFE module.

This function is to reset all the ADCs, PGAs, decimation filters and clock configuration bits. When asserted as “false”, all ADCs, PGAs and decimation filters are disabled. Clock Configuration bits are reset. When asserted as “true”, all ADCs, PGAs and decimation filters are enabled.

Parameters

- `base` – AFE peripheral base address.
- `enable` – Assert the reset command.

`static inline void AFE_Enable(AFE_Type *base, bool enable)`

Enables all configured AFE channels.

This function enables AFE and filter.

Parameters

- `base` – AFE peripheral base address.
- `enable` – Enable the AFE module or not.

`void AFE_SetChannelConfig(AFE_Type *base, uint32_t channel, const afe_channel_config_t *config)`

Configure the selected AFE channel.

This function configures the selected AFE channel.

Parameters

- base – AFE peripheral base address.
- channel – AFE channel index.
- config – Pointer to structure of “afe_channel_config_t”.

void AFE_GetDefaultChannelConfig(*afe_channel_config_t* *config)

Fills the channel configuration structure.

This function fills the afe_channel_config_t structure with default settings. Default value are:

```
config->enableHardwareTrigger    = false;
config->enableContinuousConversion = false;
config->channelMode              = kAFE_Normal;
config->decimatorOversampleRatio = kAFE_DecimatorOversampleRatio64;
config->pgaGainSelect            = kAFE_PgaGain1;
```

Parameters

- config – Pointer to structure of “afe_channel_config_t”.

uint32_t AFE_GetChannelConversionValue(AFE_Type *base, uint32_t channel)

Reads the raw conversion value.

This function returns the raw conversion value of the selected channel.

Note: The returned value could be left or right adjusted according to the AFE module configuration.

Parameters

- base – AFE peripheral base address.
- channel – AFE channel index.

Returns

Conversion value.

static inline void AFE_DoSoftwareTriggerChannel(AFE_Type *base, uint32_t mask)

Triggers the AFE conversion by software.

This function triggers the AFE conversion by executing a software command. It starts the conversion on selected channels if the software trigger option is selected for the channels.

Parameters

- base – AFE peripheral base address.
- mask – AFE channel mask software trigger. The parameter can be combination of the following source if defined:
 - kAFE_Channel0Trigger
 - kAFE_Channel1Trigger
 - kAFE_Channel2Trigger
 - kAFE_Channel3Trigger

static inline uint32_t AFE_GetChannelStatusFlags(AFE_Type *base)

Gets the AFE status flag state.

This function gets all AFE status.

Parameters

- base – AFE peripheral base address.

Returns

the mask of these status flag bits.

`void AFE_SetChannelPhaseDelayValue(AFE_Type *base, uint32_t channel, uint32_t value)`

Sets phase delays value.

This function sets the phase delays for channels. This delay is inserted before the trigger response of the decimation filters. The delay is used to provide a phase compensation between AFE channels in step of prescaled modulator clock periods.

Parameters

- base – AFE peripheral base address.
- channel – AFE channel index.
- value – delay time value.

`static inline void AFE_SetChannelPhasetDelayOk(AFE_Type *base)`

Asserts the phase delay setting.

This function should be called after all desired channel's delay registers are loaded. Values in channel's delay registers are active after calling this function and after the conversation starts.

Parameters

- base – AFE peripheral base address.

`static inline void AFE_EnableChannelInterrupts(AFE_Type *base, uint32_t mask)`

Enables AFE interrupt.

This function enables one channel interrupt.

Parameters

- base – AFE peripheral base address.
- mask – AFE channel interrupt mask. The parameter can be combination of the following source if defined:
 - kAFE_Channel0InterruptEnable
 - kAFE_Channel1InterruptEnable
 - kAFE_Channel2InterruptEnable
 - kAFE_Channel3InterruptEnable

`static inline void AFE_DisableChannelInterrupts(AFE_Type *base, uint32_t mask)`

Disables AFE interrupt.

This function disables one channel interrupt.

Parameters

- base – AFE peripheral base address.
- mask – AFE channel interrupt mask. The parameter can be combination of the following source if defined:
 - kAFE_Channel0InterruptEnable
 - kAFE_Channel1InterruptEnable
 - kAFE_Channel2InterruptEnable
 - kAFE_Channel3InterruptEnable

static inline uint32_t AFE_GetEnabledChannelInterrupts(AFE_Type *base)

Returns mask of all enabled AFE interrupts.

Parameters

- base – AFE peripheral base address.

Returns

Return the mask of these interrupt enable/disable bits.

void AFE_EnableChannelDMA(AFE_Type *base, uint32_t mask, bool enable)

Enables/Disables AFE DMA.

This function enables/disables one channel DMA request.

Parameters

- base – AFE peripheral base address.
- mask – AFE channel dma mask.
- enable – Pass true to enable interrupt, false to disable. The parameter can be combination of the following source if defined:
 - kAFE_Channel0DMAEnable
 - kAFE_Channel1DMAEnable
 - kAFE_Channel2DMAEnable
 - kAFE_Channel3DMAEnable

FSL_AFE_DRIVER_VERSION

Version 2.0.3.

enum _afe_channel_status_flag

Defines the type of status flags.

Values:

enumerator kAFE_Channel0OverflowFlag

Channel 0 previous conversion result has not been read and new data has already arrived.

enumerator kAFE_Channel1OverflowFlag

Channel 1 previous conversion result has not been read and new data has already arrived.

enumerator kAFE_Channel2OverflowFlag

Channel 2 previous conversion result has not been read and new data has already arrived.

enumerator kAFE_Channel0ReadyFlag

Channel 0 is ready to conversion.

enumerator kAFE_Channel1ReadyFlag

Channel 1 is ready to conversion.

enumerator kAFE_Channel2ReadyFlag

Channel 2 is ready to conversion.

enumerator kAFE_Channel0ConversionCompleteFlag

Channel 0 conversion is complete.

enumerator kAFE_Channel1ConversionCompleteFlag

Channel 1 conversion is complete.

enumerator kAFE_Channel2ConversionCompleteFlag

Channel 2 conversion is complete.

enumerator kAFE_Channel3OverflowFlag

Channel 3 previous conversion result has not been read and new data has already arrived.

enumerator kAFE_Channel3ReadyFlag

Channel 3 is ready to conversion.

enumerator kAFE_Channel3ConversionCompleteFlag

Channel 3 conversion is complete.

Defines AFE interrupt enable.

Values:

enumerator kAFE_Channel0InterruptEnable

Channel 0 Interrupt.

enumerator kAFE_Channel1InterruptEnable

Channel 1 Interrupt.

enumerator kAFE_Channel2InterruptEnable

Channel 2 Interrupt.

enumerator kAFE_Channel3InterruptEnable

Channel 3 Interrupt.

Defines AFE DMA enable.

Values:

enumerator kAFE_Channel0DMAEnable

Channel 0 DMA.

enumerator kAFE_Channel1DMAEnable

Channel 1 DMA.

enumerator kAFE_Channel2DMAEnable

Channel 2 DMA.

enumerator kAFE_Channel3DMAEnable

Channel 3 DMA

Defines AFE channel trigger flag.

Values:

enumerator kAFE_Channel0Trigger

Channel 0 software trigger.

enumerator kAFE_Channel1Trigger

Channel 1 software trigger.

enumerator kAFE_Channel2Trigger

Channel 2 software trigger.

enumerator kAFE_Channel3Trigger

Channel 3 software trigger.

enum `_afe_decimator_oversampling_ratio`
AFE OSR modes.

Values:

enumerator `kAFE_DecimatorOversampleRatio64`
Decimator over sample ratio is 64.

enumerator `kAFE_DecimatorOversampleRatio128`
Decimator over sample ratio is 128.

enumerator `kAFE_DecimatorOversampleRatio256`
Decimator over sample ratio is 256.

enumerator `kAFE_DecimatorOversampleRatio512`
Decimator over sample ratio is 512.

enumerator `kAFE_DecimatorOversampleRatio1024`
Decimator over sample ratio is 1024.

enumerator `kAFE_DecimatorOversampleRatio2048`
Decimator over sample ratio is 2048.

enum `_afe_result_format`
Defines the AFE result format modes.

Values:

enumerator `kAFE_ResultFormatLeft`
Left justified result format.

enumerator `kAFE_ResultFormatRight`
Right justified result format.

enum `_afe_clock_divider`
Defines the AFE clock divider modes.

Values:

enumerator `kAFE_ClockDivider1`
Clock divided by 1.

enumerator `kAFE_ClockDivider2`
Clock divided by 2.

enumerator `kAFE_ClockDivider4`
Clock divided by 4.

enumerator `kAFE_ClockDivider8`
Clock divided by 8.

enumerator `kAFE_ClockDivider16`
Clock divided by 16.

enumerator `kAFE_ClockDivider32`
Clock divided by 32.

enumerator `kAFE_ClockDivider64`
Clock divided by 64.

enumerator `kAFE_ClockDivider128`
Clock divided by 128.

enumerator kAFE_ClockDivider256

Clock divided by 256.

enum __afe_clock_source

Defines the AFE clock source modes.

Values:

enumerator kAFE_ClockSource0

Modulator clock source 0.

enumerator kAFE_ClockSource1

Modulator clock source 1.

enumerator kAFE_ClockSource2

Modulator clock source 2.

enumerator kAFE_ClockSource3

Modulator clock source 3.

enum __afe_pga_gain

Defines the PGA's values.

Values:

enumerator kAFE_PgaDisable

PGA disabled.

enumerator kAFE_PgaGain1

Input gained by 1.

enumerator kAFE_PgaGain2

Input gained by 2.

enumerator kAFE_PgaGain4

Input gained by 4.

enumerator kAFE_PgaGain8

Input gained by 8.

enumerator kAFE_PgaGain16

Input gained by 16.

enumerator kAFE_PgaGain32

Input gained by 32.

enum __afe_bypass_mode

Defines the bypass modes.

Values:

enumerator kAFE_BypassInternalClockPositiveEdge

Bypassed channel mode - internal clock selected, positive edge for registering data by the decimation filter

enumerator kAFE_BypassExternalClockPositiveEdge

Bypassed channel mode - external clock selected, positive edge for registering data by the decimation filter

enumerator kAFE_BypassInternalClockNegativeEdge

Bypassed channel mode - internal clock selected, negative edge for registering data by the decimation filter

enumerator kAFE__BypassExternalClockNegativeEdge

Bypassed channel mode - external clock selected, negative edge for registering data by the decimation filter

enumerator kAFE__BypassDisable

Normal channel mode.

typedef enum *_afe_decimator_oversampling_ratio* afe_decimator_oversample_ratio_t
AFE OSR modes.

typedef enum *_afe_result_format* afe_result_format_t
Defines the AFE result format modes.

typedef enum *_afe_clock_divider* afe_clock_divider_t
Defines the AFE clock divider modes.

typedef enum *_afe_clock_source* afe_clock_source_t
Defines the AFE clock source modes.

typedef enum *_afe_pga_gain* afe_pga_gain_t
Defines the PGA's values.

typedef enum *_afe_bypass_mode* afe_bypass_mode_t
Defines the bypass modes.

typedef struct *_afe_channel_config* afe_channel_config_t
Defines the structure to initialize the AFE channel.
This structure keeps the configuration for the AFE channel.

typedef struct *_afe_config* afe_config_t
Defines the structure to initialize the AFE module.
This structure keeps the configuration for the AFE module.

struct *_afe_channel_config*
#include <fsl_afe.h> Defines the structure to initialize the AFE channel.
This structure keeps the configuration for the AFE channel.

Public Members

bool enableHardwareTrigger
Enable triggering by hardware.

bool enableContinuousConversion
Enable continuous conversion mode.

afe_bypass_mode_t channelMode
Select if channel is in bypassed mode.

afe_pga_gain_t pgaGainSelect
Select the analog gain applied to the input signal.

afe_decimator_oversample_ratio_t decimatorOversampleRatio
Select the over sampling ration.

struct *_afe_config*
#include <fsl_afe.h> Defines the structure to initialize the AFE module.
This structure keeps the configuration for the AFE module.

Public Members

`bool enableLowPower`
Enable low power mode.

`afe_result_format_t resultFormat`
Select the result format.

`afe_clock_divider_t clockDivider`
Select the clock divider ration for the modulator clock.

`afe_clock_source_t clockSource`
Select clock source for modulator clock.

`uint8_t startupCount`
Select the start up delay of modulators.

2.3 Clock Driver

`enum __clock_name`
Clock name used to get clock frequency.

Values:

`enumerator kCLOCK_CoreSysClk`
Core/system clock

`enumerator kCLOCK_PlatClk`
Platform clock

`enumerator kCLOCK_BusClk`
Bus clock

`enumerator kCLOCK_FlashClk`
Flash clock

`enumerator kCLOCK_PllFllSelClk`
The clock after SIM[PLLFLLSEL].

`enumerator kCLOCK_Er32kClk`
External reference 32K clock (ERCLK32K)

`enumerator kCLOCK_Osc0ErClk`
OSC0 external reference clock (OSC0ERCLK)

`enumerator kCLOCK_McgFixedFreqClk`
MCG fixed frequency clock (MCGFFCLK)

`enumerator kCLOCK_McgInternalRefClk`
MCG internal reference clock (MCGIRCLK)

`enumerator kCLOCK_McgFllClk`
MCGFLLCLK

`enumerator kCLOCK_McgPll0Clk`
MCGPLL0CLK

`enumerator kCLOCK_McgExtPllClk`
EXT_PLLCLK

enumerator kCLOCK_McgPeriphClk
MCG peripheral clock (MCGPCLK)

enumerator kCLOCK_LpoClk
LPO clock

enum _clock_ip_name

Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.

Values:

enumerator kCLOCK_IpInvalid

enumerator kCLOCK_Ewm0

enumerator kCLOCK_I2c0

enumerator kCLOCK_I2c1

enumerator kCLOCK_Uart0

enumerator kCLOCK_Uart1

enumerator kCLOCK_Uart2

enumerator kCLOCK_Uart3

enumerator kCLOCK_Vref0

enumerator kCLOCK_Cmp0

enumerator kCLOCK_Cmp1

enumerator kCLOCK_Cmp2

enumerator kCLOCK_Spi0

enumerator kCLOCK_Spi1

enumerator kCLOCK_Spi2

enumerator kCLOCK_Slcd0

enumerator kCLOCK_PortA

enumerator kCLOCK_PortB

enumerator kCLOCK_PortC

enumerator kCLOCK_PortD

enumerator kCLOCK_PortE

enumerator kCLOCK_PortF

enumerator kCLOCK_PortG

enumerator kCLOCK_PortH

enumerator kCLOCK_PortI

enumerator kCLOCK_Rtc0

enumerator kCLOCK_Rtcreg

enumerator kCLOCK_Xbar
enumerator kCLOCK_Tmr0
enumerator kCLOCK_Tmr1
enumerator kCLOCK_Tmr2
enumerator kCLOCK_Tmr3
enumerator kCLOCK_Ftf0
enumerator kCLOCK_Dmamux0
enumerator kCLOCK_Rnga0
enumerator kCLOCK_Lpuart0
enumerator kCLOCK_Adc0
enumerator kCLOCK_Pit0
enumerator kCLOCK_Pit1
enumerator kCLOCK_Afe0
enumerator kCLOCK_Crc0
enumerator kCLOCK_Pdb0
enumerator kCLOCK_PortJ
enumerator kCLOCK_PortK
enumerator kCLOCK_PortL
enumerator kCLOCK_PortM
enumerator kCLOCK_Lptmr0
enumerator kCLOCK_Lptmr1
enumerator kCLOCK_Sysmpu0
enumerator kCLOCK_Dma0
enumerator kCLOCK_Cau0

enum _osc_mode

OSC work mode.

Values:

enumerator kOSC_ModeExt

Use an external clock.

enumerator kOSC_ModeOscLowPower

Oscillator low power.

enumerator kOSC_ModeOscHighGain

Oscillator high gain.

enum _osc_cap_load

Oscillator capacitor load setting.

Values:

enumerator kOSC_Cap2P

2 pF capacitor load

enumerator kOSC_Cap4P

4 pF capacitor load

enumerator kOSC_Cap8P

8 pF capacitor load

enumerator kOSC_Cap16P

16 pF capacitor load

enum __oscer_enable_mode

OSCERCLK enable mode.

Values:

enumerator kOSC_ErClkEnable

Enable.

enumerator kOSC_ErClkEnableInStop

Enable in stop mode.

enum __mcg_fl_src

MCG FLL reference clock source select.

Values:

enumerator kMCG_FllSrcExternal

External reference clock is selected

enumerator kMCG_FllSrcInternal

The slow internal reference clock is selected

enum __mcg_irc_mode

MCG internal reference clock select.

Values:

enumerator kMCG_IrcSlow

Slow internal reference clock selected

enumerator kMCG_IrcFast

Fast internal reference clock selected

enum __mcg_dmx32

MCG DCO Maximum Frequency with 32.768 kHz Reference.

Values:

enumerator kMCG_Dmx32Default

DCO has a default range of 25%

enumerator kMCG_Dmx32Fine

DCO is fine-tuned for maximum frequency with 32.768 kHz reference

enum __mcg_drs

MCG DCO range select.

Values:

enumerator kMCG_DrsLow

Low frequency range

enumerator kMCG_DrsMid
Mid frequency range

enumerator kMCG_DrsMidHigh
Mid-High frequency range

enumerator kMCG_DrsHigh
High frequency range

enum __mcg_pll_ref_src
MCG PLL reference clock select.
Values:

enumerator kMCG_PllRefRtc
Selects 32k RTC oscillator.

enumerator kMCG_PllRefIrc
Selects 32k IRC.

enumerator kMCG_PllRefFllRef
Selects FLL reference clock, the clock after FRDIV.

enum __mcg_clkout_src
MCGOUT clock source.
Values:

enumerator kMCG_ClkOutSrcOut
Output of the FLL is selected (reset default)

enumerator kMCG_ClkOutSrcInternal
Internal reference clock is selected

enumerator kMCG_ClkOutSrcExternal
External reference clock is selected

enum __mcg_atm_select
MCG Automatic Trim Machine Select.
Values:

enumerator kMCG_AtmSel32k
32 kHz Internal Reference Clock selected

enumerator kMCG_AtmSel4m
4 MHz Internal Reference Clock selected

enum __mcg_oscsel
MCG OSC Clock Select.
Values:

enumerator kMCG_OscselOsc
Selects System Oscillator (OSCCLK)

enumerator kMCG_OscselRtc
Selects 32 kHz RTC Oscillator

enum __mcg_pll_clk_select
MCG PLLCS select.
Values:

enumerator kMCG_PllClkSelPll0

PLL0 output clock is selected

enumerator kMCG_PllClkSelPll1

enum _mcg_monitor_mode

MCG clock monitor mode.

Values:

enumerator kMCG_MonitorNone

Clock monitor is disabled.

enumerator kMCG_MonitorInt

Trigger interrupt when clock lost.

enumerator kMCG_MonitorReset

System reset when clock lost.

MCG status. Enumeration _mcg_status.

Values:

enumerator kStatus_MCG_ModeUnreachable

Can't switch to target mode.

enumerator kStatus_MCG_ModeInvalid

Current mode invalid for the specific function.

enumerator kStatus_MCG_AtmBusClockInvalid

Invalid bus clock for ATM.

enumerator kStatus_MCG_AtmDesiredFreqInvalid

Invalid desired frequency for ATM.

enumerator kStatus_MCG_AtmIrcUsed

IRC is used when using ATM.

enumerator kStatus_MCG_AtmHardwareFail

Hardware fail occurs during ATM.

enumerator kStatus_MCG_SourceUsed

Can't change the clock source because it is in use.

MCG status flags. Enumeration _mcg_status_flags_t.

Values:

enumerator kMCG_Osc0LostFlag

OSC0 lost.

enumerator kMCG_Osc0InitFlag

OSC0 crystal initialized.

enumerator kMCG_RtcOscLostFlag

RTC OSC lost.

enumerator kMCG_Pll0LostFlag

PLL0 lost.

enumerator kMCG_Pll0LockFlag

PLL0 locked.

MCG internal reference clock (MCGIRCLK) enable mode definition. Enumeration `_mcg_ircclk_enable_mode`.

Values:

enumerator `kMCG_IrcclkEnable`
MCGIRCLK enable.

enumerator `kMCG_IrcclkEnableInStop`
MCGIRCLK enable in stop mode.

MCG PLL clock enable mode definition. Enumeration `_mcg_pll_enable_mode`.

Values:

enumerator `kMCG_PllEnableIndependent`
MCGPLLCLK enable independent of the MCG clock mode. Generally, the PLL is disabled in FLL modes (FEI/FBI/FEE/FBE). Setting the PLL clock enable independent, enables the PLL in the FLL modes.

enumerator `kMCG_PllEnableInStop`
MCGPLLCLK enable in STOP mode.

enum `_mcg_mode`

MCG mode definitions.

Values:

enumerator `kMCG_ModeFEI`
FEI - FLL Engaged Internal

enumerator `kMCG_ModeFBI`
FBI - FLL Bypassed Internal

enumerator `kMCG_ModeBLPI`
BLPI - Bypassed Low Power Internal

enumerator `kMCG_ModeFEE`
FEE - FLL Engaged External

enumerator `kMCG_ModeFBE`
FBE - FLL Bypassed External

enumerator `kMCG_ModeBLPE`
BLPE - Bypassed Low Power External

enumerator `kMCG_ModePBE`
PBE - PLL Bypassed External

enumerator `kMCG_ModePEE`
PEE - PLL Engaged External

enumerator `kMCG_ModePEI`
PEI - PLL Engaged Internal

enumerator `kMCG_ModePBI`
PBI - PLL Bypassed Internal

enumerator `kMCG_ModeError`
Unknown mode

typedef enum *_clock_name* clock_name_t

Clock name used to get clock frequency.

typedef enum *_clock_ip_name* clock_ip_name_t

Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.

typedef struct *_sim_clock_config* sim_clock_config_t

SIM configuration structure for clock setting.

typedef enum *_osc_mode* osc_mode_t

OSC work mode.

typedef struct *_oscer_config* oscer_config_t

OSC configuration for OSCERCLK.

typedef struct *_osc_config* osc_config_t

OSC Initialization Configuration Structure.

Defines the configuration data structure to initialize the OSC. When porting to a new board, set the following members according to the board setting:

- a. freq: The external frequency.
- b. workMode: The OSC module mode.

typedef enum *_mcg_fll_src* mcg_fll_src_t

MCG FLL reference clock source select.

typedef enum *_mcg_irc_mode* mcg_irc_mode_t

MCG internal reference clock select.

typedef enum *_mcg_dmx32* mcg_dmx32_t

MCG DCO Maximum Frequency with 32.768 kHz Reference.

typedef enum *_mcg_drs* mcg_drs_t

MCG DCO range select.

typedef enum *_mcg_pll_ref_src* mcg_pll_ref_src_t

MCG PLL reference clock select.

typedef enum *_mcg_clkout_src* mcg_clkout_src_t

MCGOUT clock source.

typedef enum *_mcg_atm_select* mcg_atm_select_t

MCG Automatic Trim Machine Select.

typedef enum *_mcg_oscsel* mcg_oscsel_t

MCG OSC Clock Select.

typedef enum *_mcg_pll_clk_select* mcg_pll_clk_select_t

MCG PLLCS select.

typedef enum *_mcg_monitor_mode* mcg_monitor_mode_t

MCG clock monitor mode.

typedef enum *_mcg_mode* mcg_mode_t

MCG mode definitions.

typedef struct *_mcg_pll_config* mcg_pll_config_t

MCG PLL configuration.

```
typedef struct _mcg_config mcg_config_t
```

MCG mode change configuration structure.

When porting to a new board, set the following members according to the board setting:

- a. `frdiv`: If the FLL uses the external reference clock, set this value to ensure that the external reference clock divided by `frdiv` is in the 31.25 kHz to 39.0625 kHz range.
- b. The PLL reference clock divider `PRDIV`: PLL reference clock frequency after `PRDIV` should be in the `FSL_FEATURE_MCG_PLL_REF_MIN` to `FSL_FEATURE_MCG_PLL_REF_MAX` range.

```
volatile uint32_t g_xtal0Freq
```

External XTAL0 (OSC0) clock frequency.

The XTAL0/EXTAL0 (OSC0) clock frequency in Hz. When the clock is set up, use the function `CLOCK_SetXtal0Freq` to set the value in the clock driver. For example, if XTAL0 is 8 MHz:

```
Set up the OSC0
CLOCK_InitOsc0(...);
Set the XTAL0 value to the clock driver.
CLOCK_SetXtal0Freq(8000000);
```

This is important for the multicore platforms where only one core needs to set up the OSC0 using the `CLOCK_InitOsc0`. All other cores need to call the `CLOCK_SetXtal0Freq` to get a valid clock frequency.

```
volatile uint32_t g_xtal32Freq
```

External XTAL32/EXTAL32/RTC_CLKIN clock frequency.

The XTAL32/EXTAL32/RTC_CLKIN clock frequency in Hz. When the clock is set up, use the function `CLOCK_SetXtal32Freq` to set the value in the clock driver.

This is important for the multicore platforms where only one core needs to set up the clock. All other cores need to call the `CLOCK_SetXtal32Freq` to get a valid clock frequency.

```
static inline void CLOCK_EnableClock(clock_ip_name_t name)
```

Enable the clock for specific IP.

Parameters

- `name` – Which clock to enable, see `clock_ip_name_t`.

```
static inline void CLOCK_DisableClock(clock_ip_name_t name)
```

Disable the clock for specific IP.

Parameters

- `name` – Which clock to disable, see `clock_ip_name_t`.

```
static inline void CLOCK_SetEr32kClock(uint32_t src)
```

Set ERCLK32K source.

Parameters

- `src` – The value to set ERCLK32K clock source.

```
static inline void CLOCK_SetLpuartClock(uint32_t src)
```

Set LPUART clock source.

Parameters

- `src` – The value to set LPUART clock source.

```
static inline void CLOCK_SetXbarClock(uint32_t src)
```

Set XBAR clock source.

Parameters

- `src` – The value to set XBAR clock source.

static inline void `CLOCK_SetAfeClkSrc(uint32_t src)`

Set the clock selection of AFECLKSEL.

Parameters

- `src` – The value to set AFECLKSEL clock source.

static inline void `CLOCK_SetPlfllSelClock(uint32_t src)`

Set PLLFLLSEL clock source.

Parameters

- `src` – The value to set PLLFLLSEL clock source.

static inline void `CLOCK_SetClkOutClock(uint32_t src)`

Set CLKOUT source.

Parameters

- `src` – The value to set CLKOUT source.

static inline void `CLOCK_SetAdcTriggerClock(uint32_t src)`

Set ADC trigger clock source.

Parameters

- `src` – The value to set ADC trigger clock source.

static inline void `CLOCK_SetOutDiv(uint32_t sysClk, uint32_t busClk, uint32_t flashClk)`

System clock divider.

Set the `SIM_CLKDIV1[OUTDIV1]`, `SIM_CLKDIV1[OUTDIV2]`, `SIM_CLKDIV1[OUTDIV3]`, `SIM_CLKDIV1[OUTDIV4]`.

Parameters

- `sysClk` – System clock divider value.
- `busClk` – Bus clock divider value.
- `flashClk` – Flash clock mode value.

uint32_t `CLOCK_GetAfeFreq(void)`

Gets the clock frequency for AFE module.

This function checks the current mode configurations in MISC_CTL register.

Returns

Clock frequency value in Hertz

uint32_t `CLOCK_GetFreq(clock_name_t clockName)`

Gets the clock frequency for a specific clock name.

This function checks the current clock configurations and then calculates the clock frequency for a specific clock name defined in `clock_name_t`. The MCG must be properly configured before using this function.

Parameters

- `clockName` – Clock names defined in `clock_name_t`

Returns

Clock frequency value in Hertz

uint32_t CLOCK_GetCoreSysClkFreq(void)

Get the core clock or system clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetPlatClkFreq(void)

Get the platform clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetBusClkFreq(void)

Get the bus clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetFlashClkFreq(void)

Get the flash clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetPllFllSelClkFreq(void)

Get the output clock frequency selected by SIM[PLL_FLLSEL].

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetEr32kClkFreq(void)

Get the external reference 32K clock frequency (ERCLK32K).

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetOsc0ErClkFreq(void)

Get the OSC0 external reference clock frequency (OSC0ERCLK).

Returns

Clock frequency in Hz.

void CLOCK_SetSimConfig(*sim_clock_config_t* const *config)

Set the clock configure in SIM module.

This function sets system layer clock settings in SIM module.

Parameters

- config – Pointer to the configure structure.

static inline void CLOCK_SetSimSafeDivs(void)

Set the system clock dividers in SIM to safe value.

The system level clocks (core clock, bus clock, flexbus clock and flash clock) must be in allowed ranges. During MCG clock mode switch, the MCG output clock changes then the system level clocks may be out of range. This function could be used before MCG mode change, to make sure system level clocks are in allowed range.

FSL_CLOCK_DRIVER_VERSION

CLOCK driver version 2.2.3.

SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY

DMAMUX_CLOCKS

Clock ip name array for DMAMUX.

RTC_CLOCKS

Clock ip name array for RTC.

SPI_CLOCKS

Clock ip name array for SPI.

SLCD_CLOCKS

Clock ip name array for SLCD.

EWM_CLOCKS

Clock ip name array for EWM.

AFE_CLOCKS

Clock ip name array for AFE.

LPUART_CLOCKS

Clock ip name array for LPUART.

ADC16_CLOCKS

Clock ip name array for ADC16.

XBAR_CLOCKS

Clock ip name array for XBAR.

SYSMPU_CLOCKS

Clock ip name array for MPU.

VREF_CLOCKS

Clock ip name array for VREF.

DMA_CLOCKS

Clock ip name array for DMA.

PORT_CLOCKS

Clock ip name array for PORT.

UART_CLOCKS

Clock ip name array for UART.

PIT_CLOCKS

Clock ip name array for PIT.

RNGA_CLOCKS

Clock ip name array for RNGA.

CRC_CLOCKS

Clock ip name array for CRC.

I2C_CLOCKS

Clock ip name array for I2C.

LPTMR_CLOCKS

Clock ip name array for LPTMR.

TMR_CLOCKS

Clock ip name array for TMR.

PDB_CLOCKS

Clock ip name array for PDB.

FTF_CLOCKS

Clock ip name array for FTF.

CMP_CLOCKS

Clock ip name array for CMP.

LPO_CLK_FREQ

LPO clock frequency.

SYS_CLK

Peripherals clock source definition.

BUS_CLK

I2C0_CLK_SRC

I2C1_CLK_SRC

SPI0_CLK_SRC

SPI1_CLK_SRC

SPI2_CLK_SRC

UART0_CLK_SRC

UART1_CLK_SRC

UART2_CLK_SRC

UART3_CLK_SRC

CLK_GATE_REG_OFFSET_SHIFT

CLK_GATE_REG_OFFSET_MASK

CLK_GATE_BIT_SHIFT_SHIFT

CLK_GATE_BIT_SHIFT_MASK

CLK_GATE_DEFINE(reg_offset, bit_shift)

CLK_GATE_ABSTRACT_REG_OFFSET(x)

CLK_GATE_ABSTRACT_BITS_SHIFT(x)

uint32_t CLOCK_GetOutClkFreq(void)

Gets the MCG output clock (MCGOUTCLK) frequency.

This function gets the MCG output clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGOUTCLK.

uint32_t CLOCK_GetFllFreq(void)

Gets the MCG FLL clock (MCGFLLCLK) frequency.

This function gets the MCG FLL clock frequency in Hz based on the current MCG register value. The FLL is enabled in FEI/FBI/FEE/FBE mode and disabled in low power state in other modes.

Returns

The frequency of MCGFLLCLK.

uint32_t CLOCK_GetInternalRefClkFreq(void)

Gets the MCG internal reference clock (MCGIRCLK) frequency.

This function gets the MCG internal reference clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGIRCLK.

uint32_t CLOCK_GetFixedFreqClkFreq(void)

Gets the MCG fixed frequency clock (MCGFFCLK) frequency.

This function gets the MCG fixed frequency clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGFFCLK.

uint32_t CLOCK_GetPll0Freq(void)

Gets the MCG PLL0 clock (MCGPLL0CLK) frequency.

This function gets the MCG PLL0 clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGPLL0CLK.

static inline void CLOCK_SetLowPowerEnable(bool enable)

Enables or disables the MCG low power.

Enabling the MCG low power disables the PLL and FLL in bypass modes. In other words, in FBE and PBE modes, enabling low power sets the MCG to BLPE mode. In FBI and PBI modes, enabling low power sets the MCG to BLPI mode. When disabling the MCG low power, the PLL or FLL are enabled based on MCG settings.

Parameters

- enable – True to enable MCG low power, false to disable MCG low power.

status_t CLOCK_SetInternalRefClkConfig(uint8_t enableMode, mcg_irc_mode_t ircs, uint8_t fcrdiv)

Configures the Internal Reference clock (MCGIRCLK).

This function sets the MCGIRCLK base on parameters. It also selects the IRC source. If the fast IRC is used, this function sets the fast IRC divider. This function also sets whether the MCGIRCLK is enabled in stop mode. Calling this function in FBI/PBI/BLPI modes may change the system clock. As a result, using the function in these modes it is not allowed.

Parameters

- enableMode – MCGIRCLK enable mode, OR'ed value of the enumeration `_mcg_ircclk_enable_mode`.
- ircs – MCGIRCLK clock source, choose fast or slow.
- fcrdiv – Fast IRC divider setting (FCRDIV).

Return values

- kStatus_MCG_SourceUsed – Because the internal reference clock is used as a clock source, the configuration should not be changed. Otherwise, a glitch occurs.
- kStatus_Success – MCGIRCLK configuration finished successfully.

status_t CLOCK_SetExternalRefClkConfig(*mcg_oscsel_t* oscsel)

Selects the MCG external reference clock.

Selects the MCG external reference clock source, changes the MCG_C7[OSCSEL], and waits for the clock source to be stable. Because the external reference clock should not be changed in FEE/FBE/BLPE/PBE/PEE modes, do not call this function in these modes.

Parameters

- oscsel – MCG external reference clock source, MCG_C7[OSCSEL].

Return values

- kStatus_MCG_SourceUsed – Because the external reference clock is used as a clock source, the configuration should not be changed. Otherwise, a glitch occurs.
- kStatus_Success – External reference clock set successfully.

static inline void CLOCK_SetFllExtRefDiv(*uint8_t* frdiv)

Set the FLL external reference clock divider value.

Sets the FLL external reference clock divider value, the register MCG_C1[FRDIV].

Parameters

- frdiv – The FLL external reference clock divider value, MCG_C1[FRDIV].

void CLOCK_EnablePll0(*mcg_pll_config_t* const *config)

Enables the PLL0 in FLL mode.

This function sets us the PLL0 in FLL mode and reconfigures the PLL0. Ensure that the PLL reference clock is enabled before calling this function and that the PLL0 is not used as a clock source. The function CLOCK_CalcPllDiv gets the correct PLL divider values.

Parameters

- config – Pointer to the configuration structure.

static inline void CLOCK_DisablePll0(void)

Disables the PLL0 in FLL mode.

This function disables the PLL0 in FLL mode. It should be used together with the CLOCK_EnablePll0.

void CLOCK_SetOsc0MonitorMode(*mcg_monitor_mode_t* mode)

Sets the OSC0 clock monitor mode.

This function sets the OSC0 clock monitor mode. See *mcg_monitor_mode_t* for details.

Parameters

- mode – Monitor mode to set.

void CLOCK_SetRtcOscMonitorMode(*mcg_monitor_mode_t* mode)

Sets the RTC OSC clock monitor mode.

This function sets the RTC OSC clock monitor mode. See *mcg_monitor_mode_t* for details.

Parameters

- mode – Monitor mode to set.

void CLOCK_SetPll0MonitorMode(*mcg_monitor_mode_t* mode)

Sets the PLL0 clock monitor mode.

This function sets the PLL0 clock monitor mode. See *mcg_monitor_mode_t* for details.

Parameters

- mode – Monitor mode to set.

uint32_t CLOCK_GetStatusFlags(void)

Gets the MCG status flags.

This function gets the MCG clock status flags. All status flags are returned as a logical OR of the enumeration refer to `_mcg_status_flags_t`. To check a specific flag, compare the return value with the flag.

Example:

To check the clock lost lock status of OSC0 and PLL0.

```
uint32_t mcgFlags;

mcgFlags = CLOCK_GetStatusFlags();

if (mcgFlags & kMCG_Osc0LostFlag)
{
    OSC0 clock lock lost. Do something.
}
if (mcgFlags & kMCG_Pll0LostFlag)
{
    PLL0 clock lock lost. Do something.
}
```

Returns

Logical OR value of the enumeration `_mcg_status_flags_t`.

void CLOCK_ClearStatusFlags(uint32_t mask)

Clears the MCG status flags.

This function clears the MCG clock lock lost status. The parameter is a logical OR value of the flags to clear. See the enumeration `_mcg_status_flags_t`.

Example:

To clear the clock lost lock status of OSC0 and PLL0.

```
CLOCK_ClearStatusFlags(kMCG_Osc0LostFlag | kMCG_Pll0LostFlag);
```

Parameters

- mask – The status flags to clear. This is a logical OR of members of the enumeration `_mcg_status_flags_t`.

static inline void OSC_SetExtRefClkConfig(OSC_Type *base, *oscer_config_t* const *config)

Configures the OSC external reference clock (OSCERCLK).

This function configures the OSC external reference clock (OSCERCLK). This is an example to enable the OSCERCLK in normal and stop modes and also set the output divider to 1:

```
oscer_config_t config =
{
    .enableMode = kOSC_ErClkEnable | kOSC_ErClkEnableInStop,
    .erclkDiv   = 1U,
};

OSC_SetExtRefClkConfig(OSC, &config);
```

Parameters

- base – OSC peripheral address.
- config – Pointer to the configuration structure.

```
static inline void OSC_SetCapLoad(OSC_Type *base, uint8_t capLoad)
```

Sets the capacitor load configuration for the oscillator.

This function sets the specified capacitors configuration for the oscillator. This should be done in the early system level initialization function call based on the system configuration.

Example:

To enable only 2 pF and 8 pF capacitor load, please use like this.
`OSC_SetCapLoad(OSC, kOSC_Cap2P | kOSC_Cap8P);`

Parameters

- base – OSC peripheral address.
- capLoad – OR'ed value for the capacitor load option, see `_osc_cap_load`.

```
void CLOCK_InitOsc0(osc_config_t const *config)
```

Initializes the OSC0.

This function initializes the OSC0 according to the board configuration.

Parameters

- config – Pointer to the OSC0 configuration structure.

```
void CLOCK_DeinitOsc0(void)
```

Deinitializes the OSC0.

This function deinitializes the OSC0.

```
static inline void CLOCK_SetXtal0Freq(uint32_t freq)
```

Sets the XTAL0 frequency based on board settings.

Parameters

- freq – The XTAL0/EXTAL0 input clock frequency in Hz.

```
static inline void CLOCK_SetXtal32Freq(uint32_t freq)
```

Sets the XTAL32/RTC_CLKIN frequency based on board settings.

Parameters

- freq – The XTAL32/EXTAL32/RTC_CLKIN input clock frequency in Hz.

```
void CLOCK_SetSlowIrcFreq(uint32_t freq)
```

Set the Slow IRC frequency based on the trimmed value.

Parameters

- freq – The Slow IRC frequency input clock frequency in Hz.

```
void CLOCK_SetFastIrcFreq(uint32_t freq)
```

Set the Fast IRC frequency based on the trimmed value.

Parameters

- freq – The Fast IRC frequency input clock frequency in Hz.

```
status_t CLOCK_TrimInternalRefClk(uint32_t extFreq, uint32_t desireFreq, uint32_t  
                                *actualFreq, mcg_atm_select_t atms)
```

Auto trims the internal reference clock.

This function trims the internal reference clock by using the external clock. If successful, it returns the `kStatus_Success` and the frequency after trimming is received in the parameter `actualFreq`. If an error occurs, the error code is returned.

Parameters

- extFreq – External clock frequency, which should be a bus clock.
- desireFreq – Frequency to trim to.
- actualFreq – Actual frequency after trimming.
- atms – Trim fast or slow internal reference clock.

Return values

- kStatus__Success – ATM success.
- kStatus__MCG__AtmBusClockInvalid – The bus clock is not in allowed range for the ATM.
- kStatus__MCG__AtmDesiredFreqInvalid – MCGIRCLK could not be trimmed to the desired frequency.
- kStatus__MCG__AtmIrcUsed – Could not trim because MCGIRCLK is used as a bus clock source.
- kStatus__MCG__AtmHardwareFail – Hardware fails while trimming.

mcg_mode_t CLOCK_GetMode(void)

Gets the current MCG mode.

This function checks the MCG registers and determines the current MCG mode.

Returns

Current MCG mode or error code; See *mcg_mode_t*.

status_t CLOCK_SetFeiMode(*mcg_dmx32_t* dmx32, *mcg_drs_t* drs, void (*fllStableDelay)(void))

Sets the MCG to FEI mode.

This function sets the MCG to FEI mode. If setting to FEI mode fails from the current mode, this function returns an error.

Note: If *dmx32* is set to *kMCG_Dmx32Fine*, the slow IRC must not be trimmed to a frequency above 32768 Hz.

Parameters

- dmx32 – DMX32 in FEI mode.
- drs – The DCO range selection.
- fllStableDelay – Delay function to ensure that the FLL is stable. Passing NULL does not cause a delay.

Return values

- kStatus__MCG__ModeUnreachable – Could not switch to the target mode.
- kStatus__Success – Switched to the target mode successfully.

status_t CLOCK_SetFeeMode(uint8_t frdiv, *mcg_dmx32_t* dmx32, *mcg_drs_t* drs, void (*fllStableDelay)(void))

Sets the MCG to FEE mode.

This function sets the MCG to FEE mode. If setting to FEE mode fails from the current mode, this function returns an error.

Parameters

- frdiv – FLL reference clock divider setting, FRDIV.
- dmx32 – DMX32 in FEE mode.

- `drs` – The DCO range selection.
- `flStableDelay` – Delay function to make sure FLL is stable. Passing NULL does not cause a delay.

Return values

- `kStatus_MCG_ModeUnreachable` – Could not switch to the target mode.
- `kStatus_Success` – Switched to the target mode successfully.

status_t CLOCK_SetFbiMode(*mcg_dmx32_t* dmx32, *mcg_drs_t* drs, void (*flStableDelay)(void))

Sets the MCG to FBI mode.

This function sets the MCG to FBI mode. If setting to FBI mode fails from the current mode, this function returns an error.

Note: If `dmx32` is set to `kMCG_Dmx32Fine`, the slow IRC must not be trimmed to frequency above 32768 Hz.

Parameters

- `dmx32` – DMX32 in FBI mode.
- `drs` – The DCO range selection.
- `flStableDelay` – Delay function to make sure FLL is stable. If the FLL is not used in FBI mode, this parameter can be NULL. Passing NULL does not cause a delay.

Return values

- `kStatus_MCG_ModeUnreachable` – Could not switch to the target mode.
- `kStatus_Success` – Switched to the target mode successfully.

status_t CLOCK_SetFbeMode(*uint8_t* frdiv, *mcg_dmx32_t* dmx32, *mcg_drs_t* drs, void (*flStableDelay)(void))

Sets the MCG to FBE mode.

This function sets the MCG to FBE mode. If setting to FBE mode fails from the current mode, this function returns an error.

Parameters

- `frdiv` – FLL reference clock divider setting, `FRDIV`.
- `dmx32` – DMX32 in FBE mode.
- `drs` – The DCO range selection.
- `flStableDelay` – Delay function to make sure FLL is stable. If the FLL is not used in FBE mode, this parameter can be NULL. Passing NULL does not cause a delay.

Return values

- `kStatus_MCG_ModeUnreachable` – Could not switch to the target mode.
- `kStatus_Success` – Switched to the target mode successfully.

status_t CLOCK_SetBlpiMode(void)

Sets the MCG to BLPI mode.

This function sets the MCG to BLPI mode. If setting to BLPI mode fails from the current mode, this function returns an error.

Return values

- `kStatus_MCG_ModeUnreachable` – Could not switch to the target mode.
- `kStatus_Success` – Switched to the target mode successfully.

status_t `CLOCK_SetBlpeMode(void)`

Sets the MCG to BLPE mode.

This function sets the MCG to BLPE mode. If setting to BLPE mode fails from the current mode, this function returns an error.

Return values

- `kStatus_MCG_ModeUnreachable` – Could not switch to the target mode.
- `kStatus_Success` – Switched to the target mode successfully.

status_t `CLOCK_SetPbeMode(mcg_pll_clk_select_t pllcs, mcg_pll_config_t const *config)`

Sets the MCG to PBE mode.

This function sets the MCG to PBE mode. If setting to PBE mode fails from the current mode, this function returns an error.

Note:

- a. The parameter `pllcs` selects the PLL. For platforms with only one PLL, the parameter `pllcs` is kept for interface compatibility.
 - b. The parameter `config` is the PLL configuration structure. On some platforms, it is possible to choose the external PLL directly, which renders the configuration structure not necessary. In this case, pass in `NULL`. For example: `CLOCK_SetPbeMode(kMCG_OscSelOsc, kMCG_PllClkSelExtPll, NULL);`
-

Parameters

- `pllcs` – The PLL selection, `PLLCS`.
- `config` – Pointer to the PLL configuration.

Return values

- `kStatus_MCG_ModeUnreachable` – Could not switch to the target mode.
- `kStatus_Success` – Switched to the target mode successfully.

status_t `CLOCK_SetPeeMode(void)`

Sets the MCG to PEE mode.

This function sets the MCG to PEE mode.

Note: This function only changes the CLKS to use the PLL/FLL output. If the PRDIV/VDIV are different than in the PBE mode, set them up in PBE mode and wait. When the clock is stable, switch to PEE mode.

Return values

- `kStatus_MCG_ModeUnreachable` – Could not switch to the target mode.
- `kStatus_Success` – Switched to the target mode successfully.

status_t `CLOCK_SetPbiMode(void)`

Sets the MCG to PBI mode.

This function sets the MCG to PBI mode.

Return values

- `kStatus_MCG_ModeUnreachable` – Could not switch to the target mode.
- `kStatus_Success` – Switched to the target mode successfully.

status_t `CLOCK_SetPeiMode(void)`

Sets the MCG to PEI mode.

This function sets the MCG to PEI mode.

Return values

- `kStatus_MCG_ModeUnreachable` – Could not switch to the target mode.
- `kStatus_Success` – Switched to the target mode successfully.

status_t `CLOCK_ExternalModeToFbeModeQuick(void)`

Switches the MCG to FBE mode from the external mode.

This function switches the MCG from external modes (PEE/PBE/BLPE/FEE) to the FBE mode quickly. The external clock is used as the system clock source and PLL is disabled. However, the FLL settings are not configured. This is a lite function with a small code size, which is useful during the mode switch. For example, to switch from PEE mode to FEI mode:

```
CLOCK_ExternalModeToFbeModeQuick();
CLOCK_SetFeiMode(...);
```

Return values

- `kStatus_Success` – Switched successfully.
- `kStatus_MCG_ModeInvalid` – If the current mode is not an external mode, do not call this function.

status_t `CLOCK_InternalModeToFbiModeQuick(void)`

Switches the MCG to FBI mode from internal modes.

This function switches the MCG from internal modes (PEI/PBI/BLPI/FEI) to the FBI mode quickly. The MCGIRCLK is used as the system clock source and PLL is disabled. However, FLL settings are not configured. This is a lite function with a small code size, which is useful during the mode switch. For example, to switch from PEI mode to FEE mode:

```
CLOCK_InternalModeToFbiModeQuick();
CLOCK_SetFeeMode(...);
```

Return values

- `kStatus_Success` – Switched successfully.
- `kStatus_MCG_ModeInvalid` – If the current mode is not an internal mode, do not call this function.

status_t `CLOCK_BootToFeiMode(mcg_dmx32_t dmx32, mcg_drs_t drs, void (*fllStableDelay)(void))`

Sets the MCG to FEI mode during system boot up.

This function sets the MCG to FEI mode from the reset mode. It can also be used to set up MCG during system boot up.

Note: If `dmx32` is set to `kMCG_Dmx32Fine`, the slow IRC must not be trimmed to frequency above 32768 Hz.

Parameters

- `dmx32` – DMX32 in FEI mode.

- `drs` – The DCO range selection.
- `flStableDelay` – Delay function to ensure that the FLL is stable.

Return values

- `kStatus_MCG_ModeUnreachable` – Could not switch to the target mode.
- `kStatus_Success` – Switched to the target mode successfully.

status_t CLOCK_BootToFeeMode(*mcg_oscsel_t* oscsel, *uint8_t* frdiv, *mcg_dm32_t* dm32, *mcg_drs_t* drs, void (*flStableDelay)(void))

Sets the MCG to FEE mode during system bootup.

This function sets MCG to FEE mode from the reset mode. It can also be used to set up the MCG during system boot up.

Parameters

- `oscsel` – OSC clock select, OSCSEL.
- `frdiv` – FLL reference clock divider setting, FRDIV.
- `dm32` – DMX32 in FEE mode.
- `drs` – The DCO range selection.
- `flStableDelay` – Delay function to ensure that the FLL is stable.

Return values

- `kStatus_MCG_ModeUnreachable` – Could not switch to the target mode.
- `kStatus_Success` – Switched to the target mode successfully.

status_t CLOCK_BootToBlpiMode(*uint8_t* fcrdiv, *mcg_irc_mode_t* ircs, *uint8_t* ircEnableMode)

Sets the MCG to BLPI mode during system boot up.

This function sets the MCG to BLPI mode from the reset mode. It can also be used to set up the MCG during system boot up.

Parameters

- `fcrdiv` – Fast IRC divider, FCRDIV.
- `ircs` – The internal reference clock to select, IRCS.
- `ircEnableMode` – The MCGIRCLK enable mode, OR'ed value of the enumeration `_mcg_ircclk_enable_mode`.

Return values

- `kStatus_MCG_SourceUsed` – Could not change MCGIRCLK setting.
- `kStatus_Success` – Switched to the target mode successfully.

status_t CLOCK_BootToBlpeMode(*mcg_oscsel_t* oscsel)

Sets the MCG to BLPE mode during system boot up.

This function sets the MCG to BLPE mode from the reset mode. It can also be used to set up the MCG during system boot up.

Parameters

- `oscsel` – OSC clock select, MCG_C7[OSCSEL].

Return values

- `kStatus_MCG_ModeUnreachable` – Could not switch to the target mode.
- `kStatus_Success` – Switched to the target mode successfully.

status_t CLOCK_BootToPeeMode(*mcg_oscsel_t* oscsel, *mcg_pll_clk_select_t* pllcs, *mcg_pll_config_t* const *config)

Sets the MCG to PEE mode during system boot up.

This function sets the MCG to PEE mode from reset mode. It can also be used to set up the MCG during system boot up.

Parameters

- oscsel – OSC clock select, MCG_C7[OSCSSEL].
- pllcs – The PLL selection, PLLCS.
- config – Pointer to the PLL configuration.

Return values

- kStatus_MCG_ModeUnreachable – Could not switch to the target mode.
- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_BootToPeiMode(void)

Sets the MCG to PEI mode during system boot up.

This function sets the MCG to PEI mode from the reset mode. It can be used to set up the MCG during system boot up.

Return values

- kStatus_MCG_ModeUnreachable – Could not switch to the target mode.
- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_SetMcgConfig(*mcg_config_t* const *config)

Sets the MCG to a target mode.

This function sets MCG to a target mode defined by the configuration structure. If switching to the target mode fails, this function chooses the correct path.

Note: If the external clock is used in the target mode, ensure that it is enabled. For example, if the OSC0 is used, set up OSC0 correctly before calling this function.

Parameters

- config – Pointer to the target MCG mode configuration structure.

Returns

Return kStatus_Success if switched successfully; Otherwise, it returns an error code `_mcg_status`.

uint8_t pllFllSel

PLL/FLL/IRC48M selection.

uint8_t er32kSrc

ERCLK32K source selection.

uint32_t clkdiv1

SIM_CLKDIV1.

uint8_t enableMode

OSCERCLK enable mode. OR'ed value of `_oscer_enable_mode`.

uint32_t freq

External clock frequency.

`uint8_t capLoad`

Capacitor load setting.

`osc_mode_t workMode`

OSC work mode setting.

`oscer_config_t oscerConfig`

Configuration for OSCERCLK.

`uint8_t enableMode`

Enable mode. OR'ed value of enumeration `_mcg_pll_enable_mode`.

`mcg_pll_ref_src_t refSrc`

PLL reference clock source.

`uint8_t frdiv`

FLL reference clock divider.

`mcg_mode_t mcgMode`

MCG mode.

`uint8_t irclkEnableMode`

MCGIRCLK enable mode.

`mcg_irc_mode_t ircs`

Source, MCG_C2[IRCS].

`uint8_t fcrdiv`

Divider, MCG_SC[FCRDIV].

`uint8_t frdiv`

Divider MCG_C1[FRDIV].

`mcg_drs_t drs`

DCO range MCG_C4[DRST_DRS].

`mcg_dm32_t dm32`

MCG_C4[DMX32].

`mcg_oscsel_t oscsel`

OSC select MCG_C7[OSCSSEL].

`mcg_pll_config_t pll0Config`

MCGPLL0CLK configuration.

`MCG_CONFIG_CHECK_PARAM`

Configures whether to check a parameter in a function.

Some MCG settings must be changed with conditions, for example:

- a. MCGIRCLK settings, such as the source, divider, and the trim value should not change when MCGIRCLK is used as a system clock source.
- b. MCG_C7[OSCSSEL] should not be changed when the external reference clock is used as a system clock source. For example, in FBE/BLPE/PBE modes.
- c. The users should only switch between the supported clock modes.

MCG functions check the parameter and MCG status before setting, if not allowed to change, the functions return error. The parameter checking increases code size, if code size is a critical requirement, change `MCG_CONFIG_CHECK_PARAM` to 0 to disable parameter checking.

FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL

Configure whether driver controls clock.

When set to 0, peripheral drivers will enable clock in initialize function and disable clock in de-initialize function. When set to 1, peripheral driver will not control the clock, application could control the clock out of the driver.

Note: All drivers share this feature switcher. If it is set to 1, application should handle clock enable and disable for all drivers.

struct __sim_clock_config

#include <fsl_clock.h> SIM configuration structure for clock setting.

struct __oscer_config

#include <fsl_clock.h> OSC configuration for OSCERCLK.

struct __osc_config

#include <fsl_clock.h> OSC Initialization Configuration Structure.

Defines the configuration data structure to initialize the OSC. When porting to a new board, set the following members according to the board setting:

- a. freq: The external frequency.
- b. workMode: The OSC module mode.

struct __mcg_pll_config

#include <fsl_clock.h> MCG PLL configuration.

struct __mcg_config

#include <fsl_clock.h> MCG mode change configuration structure.

When porting to a new board, set the following members according to the board setting:

- a. frdiv: If the FLL uses the external reference clock, set this value to ensure that the external reference clock divided by frdiv is in the 31.25 kHz to 39.0625 kHz range.
- b. The PLL reference clock divider PRDIV: PLL reference clock frequency after PRDIV should be in the FSL_FEATURE_MCG_PLL_REF_MIN to FSL_FEATURE_MCG_PLL_REF_MAX range.

2.4 CMP: Analog Comparator Driver

void CMP_Init(CMP_Type *base, const *cmp_config_t* *config)

Initializes the CMP.

This function initializes the CMP module. The operations included are as follows.

- Enabling the clock for CMP module.
- Configuring the comparator.
- Enabling the CMP module. Note that for some devices, multiple CMP instances share the same clock gate. In this case, to enable the clock for any instance enables all CMPs. See the appropriate MCU reference manual for the clock assignment of the CMP.

Parameters

- base – CMP peripheral base address.
- config – Pointer to the configuration structure.

```
void CMP_Deinit(CMP_Type *base)
```

De-initializes the CMP module.

This function de-initializes the CMP module. The operations included are as follows.

- Disabling the CMP module.
- Disabling the clock for CMP module.

This function disables the clock for the CMP. Note that for some devices, multiple CMP instances share the same clock gate. In this case, before disabling the clock for the CMP, ensure that all the CMP instances are not used.

Parameters

- base – CMP peripheral base address.

```
static inline void CMP_Enable(CMP_Type *base, bool enable)
```

Enables/disables the CMP module.

Parameters

- base – CMP peripheral base address.
- enable – Enables or disables the module.

```
void CMP_GetDefaultConfig(cmp_config_t *config)
```

Initializes the CMP user configuration structure.

This function initializes the user configuration structure to these default values.

```
config->enableCmp      = true;
config->hysteresisMode = kCMP_HysteresisLevel0;
config->enableHighSpeed = false;
config->enableInvertOutput = false;
config->useUnfilteredOutput = false;
config->enablePinOut     = false;
config->enableTriggerMode = false;
```

Parameters

- config – Pointer to the configuration structure.

```
void CMP_SetInputChannels(CMP_Type *base, uint8_t positiveChannel, uint8_t
                          negativeChannel)
```

Sets the input channels for the comparator.

This function sets the input channels for the comparator. Note that two input channels cannot be set the same way in the application. When the user selects the same input from the analog mux to the positive and negative port, the comparator is disabled automatically.

Parameters

- base – CMP peripheral base address.
- positiveChannel – Positive side input channel number. Available range is 0-7.
- negativeChannel – Negative side input channel number. Available range is 0-7.

```
void CMP_EnableDMA(CMP_Type *base, bool enable)
```

Enables/disables the DMA request for rising/falling events.

This function enables/disables the DMA request for rising/falling events. Either event triggers the generation of the DMA request from CMP if the DMA feature is enabled. Both events are ignored for generating the DMA request from the CMP if the DMA is disabled.

Parameters

- base – CMP peripheral base address.
- enable – Enables or disables the feature.

static inline void CMP__EnableWindowMode(CMP_Type *base, bool enable)

Enables/disables the window mode.

Parameters

- base – CMP peripheral base address.
- enable – Enables or disables the feature.

static inline void CMP__EnablePassThroughMode(CMP_Type *base, bool enable)

Enables/disables the pass through mode.

Parameters

- base – CMP peripheral base address.
- enable – Enables or disables the feature.

void CMP__SetFilterConfig(CMP_Type *base, const *cmp_filter_config_t* *config)

Configures the filter.

Parameters

- base – CMP peripheral base address.
- config – Pointer to the configuration structure.

void CMP__SetDACConfig(CMP_Type *base, const *cmp_dac_config_t* *config)

Configures the internal DAC.

Parameters

- base – CMP peripheral base address.
- config – Pointer to the configuration structure. “NULL” disables the feature.

void CMP__EnableInterrupts(CMP_Type *base, uint32_t mask)

Enables the interrupts.

Parameters

- base – CMP peripheral base address.
- mask – Mask value for interrupts. See “_cmp_interrupt_enable”.

void CMP__DisableInterrupts(CMP_Type *base, uint32_t mask)

Disables the interrupts.

Parameters

- base – CMP peripheral base address.
- mask – Mask value for interrupts. See “_cmp_interrupt_enable”.

uint32_t CMP__GetStatusFlags(CMP_Type *base)

Gets the status flags.

Parameters

- base – CMP peripheral base address.

Returns

Mask value for the asserted flags. See “_cmp_status_flags”.

`void CMP_ClearStatusFlags(CMP_Type *base, uint32_t mask)`

Clears the status flags.

Parameters

- `base` – CMP peripheral base address.
- `mask` – Mask value for the flags. See “`_cmp_status_flags`”.

`FSL_CMP_DRIVER_VERSION`

CMP driver version 2.0.3.

`enum _cmp_interrupt_enable`

Interrupt enable/disable mask.

Values:

`enumerator kCMP_OutputRisingInterruptEnable`

Comparator interrupt enable rising.

`enumerator kCMP_OutputFallingInterruptEnable`

Comparator interrupt enable falling.

`enum _cmp_status_flags`

Status flags' mask.

Values:

`enumerator kCMP_OutputRisingEventFlag`

Rising-edge on the comparison output has occurred.

`enumerator kCMP_OutputFallingEventFlag`

Falling-edge on the comparison output has occurred.

`enumerator kCMP_OutputAssertEventFlag`

Return the current value of the analog comparator output.

`enum _cmp_hysteresis_mode`

CMP Hysteresis mode.

Values:

`enumerator kCMP_HysteresisLevel0`

Hysteresis level 0.

`enumerator kCMP_HysteresisLevel1`

Hysteresis level 1.

`enumerator kCMP_HysteresisLevel2`

Hysteresis level 2.

`enumerator kCMP_HysteresisLevel3`

Hysteresis level 3.

`enum _cmp_reference_voltage_source`

CMP Voltage Reference source.

Values:

`enumerator kCMP_VrefSourceVin1`

Vin1 is selected as a resistor ladder network supply reference Vin.

`enumerator kCMP_VrefSourceVin2`

Vin2 is selected as a resistor ladder network supply reference Vin.

```
typedef enum _cmp_hysteresis_mode cmp_hysteresis_mode_t
    CMP Hysteresis mode.

typedef enum _cmp_reference_voltage_source cmp_reference_voltage_source_t
    CMP Voltage Reference source.

typedef struct _cmp_config cmp_config_t
    Configures the comparator.

typedef struct _cmp_filter_config cmp_filter_config_t
    Configures the filter.

typedef struct _cmp_dac_config cmp_dac_config_t
    Configures the internal DAC.

struct _cmp_config
    #include <fsl_cmp.h> Configures the comparator.
```

Public Members

```
bool enableCmp
    Enable the CMP module.

cmp_hysteresis_mode_t hysteresisMode
    CMP Hysteresis mode.

bool enableHighSpeed
    Enable High-speed (HS) comparison mode.

bool enableInvertOutput
    Enable the inverted comparator output.

bool useUnfilteredOutput
    Set the compare output(COUT) to equal COUTA(true) or COUT(false).

bool enablePinOut
    The comparator output is available on the associated pin.

bool enableTriggerMode
    Enable the trigger mode.

struct _cmp_filter_config
    #include <fsl_cmp.h> Configures the filter.
```

Public Members

```
bool enableSample
    Using the external SAMPLE as a sampling clock input or using a divided bus clock.

uint8_t filterCount
    Filter Sample Count. Available range is 1-7; 0 disables the filter.

uint8_t filterPeriod
    Filter Sample Period. The divider to the bus clock. Available range is 0-255.

struct _cmp_dac_config
    #include <fsl_cmp.h> Configures the internal DAC.
```

Public Members

cmp_reference_voltage_source_t referenceVoltageSource

Supply voltage reference source.

uint8_t DACValue

Value for the DAC Output Voltage. Available range is 0-63.

2.5 CRC: Cyclic Redundancy Check Driver

FSL_CRC_DRIVER_VERSION

CRC driver version. Version 2.0.5.

Current version: 2.0.5

Change log:

- Version 2.0.5
 - Fix CERT-C issue with boolean-to-unsigned integer conversion.
- Version 2.0.4
 - Release peripheral from reset if necessary in init function.
- Version 2.0.3
 - Fix MISRA issues
- Version 2.0.2
 - Fix MISRA issues
- Version 2.0.1
 - move DATA and DATALL macro definition from header file to source file

enum _crc_bits

CRC bit width.

Values:

enumerator *kCrcBits16*

Generate 16-bit CRC code

enumerator *kCrcBits32*

Generate 32-bit CRC code

enum _crc_result

CRC result type.

Values:

enumerator *kCrcFinalChecksum*

CRC data register read value is the final checksum. Reflect out and final xor protocol features are applied.

enumerator *kCrcIntermediateChecksum*

CRC data register read value is intermediate checksum (raw value). Reflect out and final xor protocol feature are not applied. Intermediate checksum can be used as a seed for *CRC_Init()* to continue adding data to this checksum.

typedef enum *_crc_bits* crc_bits_t

CRC bit width.

typedef enum *_crc_result* crc_result_t

CRC result type.

typedef struct *_crc_config* crc_config_t

CRC protocol configuration.

This structure holds the configuration for the CRC protocol.

void CRC_Init(CRC_Type *base, const *crc_config_t* *config)

Enables and configures the CRC peripheral module.

This function enables the clock gate in the SIM module for the CRC peripheral. It also configures the CRC module and starts a checksum computation by writing the seed.

Parameters

- base – CRC peripheral address.
- config – CRC module configuration structure.

static inline void CRC_Deinit(CRC_Type *base)

Disables the CRC peripheral module.

This function disables the clock gate in the SIM module for the CRC peripheral.

Parameters

- base – CRC peripheral address.

void CRC_GetDefaultConfig(*crc_config_t* *config)

Loads default values to the CRC protocol configuration structure.

Loads default values to the CRC protocol configuration structure. The default values are as follows.

```
config->polynomial = 0x1021;
config->seed = 0xFFFF;
config->reflectIn = false;
config->reflectOut = false;
config->complementChecksum = false;
config->crcBits = kCrcBits16;
config->crcResult = kCrcFinalChecksum;
```

Parameters

- config – CRC protocol configuration structure.

void CRC_WriteData(CRC_Type *base, const uint8_t *data, size_t dataSize)

Writes data to the CRC module.

Writes input data buffer bytes to the CRC data register. The configured type of transpose is applied.

Parameters

- base – CRC peripheral address.
- data – Input data stream, MSByte in data[0].
- dataSize – Size in bytes of the input data buffer.

uint32_t CRC_Get32bitResult(CRC_Type *base)

Reads the 32-bit checksum from the CRC module.

Reads the CRC data register (either an intermediate or the final checksum). The configured type of transpose and complement is applied.

Parameters

- base – CRC peripheral address.

Returns

An intermediate or the final 32-bit checksum, after configured transpose and complement operations.

uint16_t CRC_Get16bitResult(CRC_Type *base)

Reads a 16-bit checksum from the CRC module.

Reads the CRC data register (either an intermediate or the final checksum). The configured type of transpose and complement is applied.

Parameters

- base – CRC peripheral address.

Returns

An intermediate or the final 16-bit checksum, after configured transpose and complement operations.

CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT

Default configuration structure filled by CRC_GetDefaultConfig(). Use CRC16-CCIT-FALSE as default.

struct _crc_config

#include <fsl_crc.h> CRC protocol configuration.

This structure holds the configuration for the CRC protocol.

Public Members

uint32_t polynomial

CRC Polynomial, MSBit first. Example polynomial: $0x1021 = 1\_0000\_0010\_0001 = x^{12} + x^5 + 1$

uint32_t seed

Starting checksum value

bool reflectIn

Reflect bits on input.

bool reflectOut

Reflect bits on output.

bool complementChecksum

True if the result shall be complement of the actual checksum.

crc_bits_t crcBits

Selects 16- or 32- bit CRC protocol.

crc_result_t crcResult

Selects final or intermediate checksum return from CRC_Get16bitResult() or CRC_Get32bitResult()

2.6 DMA: Direct Memory Access Controller Driver

void DMA_Init(DMA_Type *base)

Initializes the DMA peripheral.

This function ungates the DMA clock.

Parameters

- base – DMA peripheral base address.

void DMA_Deinit(DMA_Type *base)

Deinitializes the DMA peripheral.

This function gates the DMA clock.

Parameters

- base – DMA peripheral base address.

void DMA_ResetChannel(DMA_Type *base, uint32_t channel)

Resets the DMA channel.

Sets all register values to reset values and enables the cycle steal and auto stop channel request features.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

void DMA_SetTransferConfig(DMA_Type *base, uint32_t channel, const *dma_transfer_config_t* *config)

Configures the DMA transfer attribute.

This function configures the transfer attribute including the source address, destination address, transfer size, and so on. This example shows how to set up the *dma_transfer_config_t* parameters and how to call the *DMA_ConfigBasicTransfer* function.

```
dma_transfer_config_t transferConfig;
memset(&transferConfig, 0, sizeof(transferConfig));
transferConfig.srcAddr = (uint32_t)srcAddr;
transferConfig.destAddr = (uint32_t)destAddr;
transferConfig.enableSrcIncrement = true;
transferConfig.enableDestIncrement = true;
transferConfig.srcSize = kDMA_Transfersize32bits;
transferConfig.destSize = kDMA_Transfersize32bits;
transferConfig.transferSize = sizeof(uint32_t) * BUFF_LENGTH;
DMA_SetTransferConfig(DMA0, 0, &transferConfig);
```

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- config – Pointer to the DMA transfer configuration structure.

void DMA_SetChannelLinkConfig(DMA_Type *base, uint32_t channel, const *dma_channel_link_config_t* *config)

Configures the DMA channel link feature.

This function allows DMA channels to have their transfers linked. The current DMA channel triggers a DMA request to the linked channels (LCH1 or LCH2) depending on the channel link type. Perform a link to channel LCH1 after each cycle-steal

transfer followed by a link to LCH2 after the BCR decrements to 0 if the type is kDMA_ChannelLinkChannel1AndChannel2. Perform a link to LCH1 after each cycle-steal transfer if the type is kDMA_ChannelLinkChannel1. Perform a link to LCH1 after the BCR decrements to 0 if the type is kDMA_ChannelLinkChannel1AfterBCR0.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- config – Pointer to the channel link configuration structure.

```
static inline void DMA_SetSourceAddress(DMA_Type *base, uint32_t channel, uint32_t srcAddr)
```

Sets the DMA source address for the DMA transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- srcAddr – DMA source address.

```
static inline void DMA_SetDestinationAddress(DMA_Type *base, uint32_t channel, uint32_t destAddr)
```

Sets the DMA destination address for the DMA transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- destAddr – DMA destination address.

```
static inline void DMA_SetTransferSize(DMA_Type *base, uint32_t channel, uint32_t size)
```

Sets the DMA transfer size for the DMA transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- size – The number of bytes to be transferred.

```
void DMA_SetModulo(DMA_Type *base, uint32_t channel, dma_modulo_t srcModulo, dma_modulo_t destModulo)
```

Sets the DMA modulo for the DMA transfer.

This function defines a specific address range specified to be the value after (SAR + SSIZE)/(DAR + DSIZE) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- srcModulo – source address modulo.
- destModulo – destination address modulo.

```
static inline void DMA_EnableCycleSteal(DMA_Type *base, uint32_t channel, bool enable)
```

Enables the DMA cycle steal for the DMA transfer.

If the cycle steal feature is enabled (true), the DMA controller forces a single read/write transfer per request, or it continuously makes read/write transfers until the BCR decrements to 0.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- enable – The command for enable (true) or disable (false).

static inline void DMA_EnableAutoAlign(DMA_Type *base, uint32_t channel, bool enable)

Enables the DMA auto align for the DMA transfer.

If the auto align feature is enabled (true), the appropriate address register increments regardless of DINC or SINC.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- enable – The command for enable (true) or disable (false).

static inline void DMA_EnableAsyncRequest(DMA_Type *base, uint32_t channel, bool enable)

Enables the DMA async request for the DMA transfer.

If the async request feature is enabled (true), the DMA supports asynchronous DREQs while the MCU is in stop mode.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- enable – The command for enable (true) or disable (false).

static inline void DMA_EnableInterrupts(DMA_Type *base, uint32_t channel)

Enables an interrupt for the DMA transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_DisableInterrupts(DMA_Type *base, uint32_t channel)

Disables an interrupt for the DMA transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_EnableChannelRequest(DMA_Type *base, uint32_t channel)

Enables the DMA hardware channel request.

Parameters

- base – DMA peripheral base address.
- channel – The DMA channel number.

static inline void DMA_DisableChannelRequest(DMA_Type *base, uint32_t channel)

Disables the DMA hardware channel request.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

```
static inline void DMA_TriggerChannelStart(DMA_Type *base, uint32_t channel)
```

Starts the DMA transfer with a software trigger.

This function starts only one read/write iteration.

Parameters

- base – DMA peripheral base address.
- channel – The DMA channel number.

```
static inline void DMA_EnableAutoStopRequest(DMA_Type *base, uint32_t channel, bool enable)
```

Starts the DMA enable/disable auto disable request.

Parameters

- base – DMA peripheral base address.
- channel – The DMA channel number.
- enable – true is enable, false is disable.

```
static inline uint32_t DMA_GetRemainingBytes(DMA_Type *base, uint32_t channel)
```

Gets the remaining bytes of the current DMA transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

Returns

The number of bytes which have not been transferred yet.

```
static inline uint32_t DMA_GetChannelStatusFlags(DMA_Type *base, uint32_t channel)
```

Gets the DMA channel status flags.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

Returns

The mask of the channel status. Use the `_dma_channel_status_flags` type to decode the return 32 bit variables.

```
static inline void DMA_ClearChannelStatusFlags(DMA_Type *base, uint32_t channel, uint32_t mask)
```

Clears the DMA channel status flags.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- mask – The mask of the channel status to be cleared. Use the defined `_dma_channel_status_flags` type.

```
void DMA_CreateHandle(dma_handle_t *handle, DMA_Type *base, uint32_t channel)
```

Creates the DMA handle.

This function is called first if using the transactional API for the DMA. This function initializes the internal state of the DMA handle.

Parameters

- handle – DMA handle pointer. The DMA handle stores callback function and parameters.

- base – DMA peripheral base address.
- channel – DMA channel number.

void DMA_SetCallback(*dma_handle_t* *handle, *dma_callback* callback, void *userData)

Sets the DMA callback function.

This callback is called in the DMA IRQ handler. Use the callback to do something after the current transfer complete.

Parameters

- handle – DMA handle pointer.
- callback – DMA callback function pointer.
- userData – Parameter for callback function. If it is not needed, just set to NULL.

void DMA_PrepareTransferConfig(*dma_transfer_config_t* *config, void *srcAddr, uint32_t srcWidth, void *destAddr, uint32_t destWidth, uint32_t transferBytes, *dma_addr_increment_t* srcIncrement, *dma_addr_increment_t* destIncrement)

Prepares the DMA transfer configuration structure.

This function prepares the transfer configuration structure according to the user input. The difference between this function and DMA_PrepareTransfer is that this function expose the address increment parameter to application, but in DMA_PrepareTransfer, only parts of the address increment option can be selected by *dma_transfer_type_t*.

Parameters

- config – Pointer to the user configuration structure of type *dma_transfer_config_t*.
- srcAddr – DMA transfer source address.
- srcWidth – DMA transfer source address width (byte).
- destAddr – DMA transfer destination address.
- destWidth – DMA transfer destination address width (byte).
- transferBytes – DMA transfer bytes to be transferred.
- srcIncrement – source address increment type.
- destIncrement – dest address increment type.

void DMA_PrepareTransfer(*dma_transfer_config_t* *config, void *srcAddr, uint32_t srcWidth, void *destAddr, uint32_t destWidth, uint32_t transferBytes, *dma_transfer_type_t* type)

Prepares the DMA transfer configuration structure.

This function prepares the transfer configuration structure according to the user input.

Parameters

- config – Pointer to the user configuration structure of type *dma_transfer_config_t*.
- srcAddr – DMA transfer source address.
- srcWidth – DMA transfer source address width (byte).
- destAddr – DMA transfer destination address.
- destWidth – DMA transfer destination address width (byte).
- transferBytes – DMA transfer bytes to be transferred.

- type – DMA transfer type.

status_t DMA_SubmitTransfer(*dma_handle_t* *handle, const *dma_transfer_config_t* *config, uint32_t options)

Submits the DMA transfer request.

This function submits the DMA transfer request according to the transfer configuration structure.

Note: This function can't process multi transfer request.

Parameters

- handle – DMA handle pointer.
- config – Pointer to DMA transfer configuration structure.
- options – Additional configurations for transfer. Use the defined *dma_transfer_options_t* type.

Return values

- *kStatus_DMA_Success* – It indicates that the DMA submit transfer request succeeded.
- *kStatus_DMA_Busy* – It indicates that the DMA is busy. Submit transfer request is not allowed.

static inline void DMA_StartTransfer(*dma_handle_t* *handle)

DMA starts a transfer.

This function enables the channel request. Call this function after submitting a transfer request.

Parameters

- handle – DMA handle pointer.

Return values

- *kStatus_DMA_Success* – It indicates that the DMA start transfer succeed.
- *kStatus_DMA_Busy* – It indicates that the DMA has started a transfer.

static inline void DMA_StopTransfer(*dma_handle_t* *handle)

DMA stops a transfer.

This function disables the channel request to stop a DMA transfer. The transfer can be resumed by calling the *DMA_StartTransfer*.

Parameters

- handle – DMA handle pointer.

void DMA_AbortTransfer(*dma_handle_t* *handle)

DMA aborts a transfer.

This function disables the channel request and clears all status bits. Submit another transfer after calling this API.

Parameters

- handle – DMA handle pointer.

void DMA_HandleIRQ(*dma_handle_t* *handle)

DMA IRQ handler for current transfer complete.

This function clears the channel interrupt flag and calls the callback function if it is not NULL.

Parameters

- handle – DMA handle pointer.

FSL_DMA_DRIVER_VERSION

DMA driver version 2.1.3.

_dma_channel_status_flags status flag for the DMA driver.

Values:

enumerator kDMA_TransactionsBCRFlag

Contains the number of bytes yet to be transferred for a given block

enumerator kDMA_TransactionsDoneFlag

Transactions Done

enumerator kDMA_TransactionsBusyFlag

Transactions Busy

enumerator kDMA_TransactionsRequestFlag

Transactions Request

enumerator kDMA_BusErrorOnDestinationFlag

Bus Error on Destination

enumerator kDMA_BusErrorOnSourceFlag

Bus Error on Source

enumerator kDMA_ConfigurationErrorFlag

Configuration Error

enum _dma_transfer_size

DMA transfer size type.

Values:

enumerator kDMA_Transfersize32bits

32 bits are transferred for every read/write

enumerator kDMA_Transfersize8bits

8 bits are transferred for every read/write

enumerator kDMA_Transfersize16bits

16b its are transferred for every read/write

enum _dma_modulo

Configuration type for the DMA modulo.

Values:

enumerator kDMA_ModuloDisable

Buffer disabled

enumerator kDMA_Modulo16Bytes

Circular buffer size is 16 bytes.

enumerator kDMA_Modulo32Bytes
Circular buffer size is 32 bytes.

enumerator kDMA_Modulo64Bytes
Circular buffer size is 64 bytes.

enumerator kDMA_Modulo128Bytes
Circular buffer size is 128 bytes.

enumerator kDMA_Modulo256Bytes
Circular buffer size is 256 bytes.

enumerator kDMA_Modulo512Bytes
Circular buffer size is 512 bytes.

enumerator kDMA_Modulo1KBytes
Circular buffer size is 1 KB.

enumerator kDMA_Modulo2KBytes
Circular buffer size is 2 KB.

enumerator kDMA_Modulo4KBytes
Circular buffer size is 4 KB.

enumerator kDMA_Modulo8KBytes
Circular buffer size is 8 KB.

enumerator kDMA_Modulo16KBytes
Circular buffer size is 16 KB.

enumerator kDMA_Modulo32KBytes
Circular buffer size is 32 KB.

enumerator kDMA_Modulo64KBytes
Circular buffer size is 64 KB.

enumerator kDMA_Modulo128KBytes
Circular buffer size is 128 KB.

enumerator kDMA_Modulo256KBytes
Circular buffer size is 256 KB.

enum _dma_channel_link_type
DMA channel link type.

Values:

enumerator kDMA_ChannelLinkDisable
No channel link.

enumerator kDMA_ChannelLinkChannel1AndChannel2
Perform a link to channel LCH1 after each cycle-steal transfer. followed by a link to LCH2 after the BCR decrements to 0.

enumerator kDMA_ChannelLinkChannel1
Perform a link to LCH1 after each cycle-steal transfer.

enumerator kDMA_ChannelLinkChannel1AfterBCR0
Perform a link to LCH1 after the BCR decrements.

enum _dma_transfer_type
DMA transfer type.

Values:

enumerator kDMA_MemoryToMemory

Memory to Memory transfer.

enumerator kDMA_PeripheralToMemory

Peripheral to Memory transfer.

enumerator kDMA_MemoryToPeripheral

Memory to Peripheral transfer.

enum _dma_transfer_options

DMA transfer options.

Values:

enumerator kDMA_NoOptions

Transfer without options.

enumerator kDMA_EnableInterrupt

Enable interrupt while transfer complete.

enum _dma_addr_increment

dma addre increment type

Values:

enumerator kDMA_AddrNoIncrement

Transfer address not increment.

enumerator kDMA_AddrIncrementPerTransferWidth

Transfer address increment per transfer width

_dma_transfer_status DMA transfer status

Values:

enumerator kStatus_DMA_Busy

DMA is busy.

typedef enum _dma_transfer_size dma_transfer_size_t

DMA transfer size type.

typedef enum _dma_modulo dma_modulo_t

Configuration type for the DMA modulo.

typedef enum _dma_channel_link_type dma_channel_link_type_t

DMA channel link type.

typedef enum _dma_transfer_type dma_transfer_type_t

DMA transfer type.

typedef enum _dma_transfer_options dma_transfer_options_t

DMA transfer options.

typedef enum _dma_addr_increment dma_addr_increment_t

dma addre increment type

typedef struct _dma_transfer_config dma_transfer_config_t

DMA transfer configuration structure.

typedef struct _dma_channel_link_config dma_channel_link_config_t

DMA transfer configuration structure.

```
typedef void (*dma_callback)(struct_dma_handle *handle, void *userData)
```

Callback function prototype for the DMA driver.

```
typedef struct_dma_handle dma_handle_t
```

DMA DMA handle structure.

```
struct_dma_transfer_config
```

#include <fsl_dma.h> DMA transfer configuration structure.

Public Members

uint32_t srcAddr

DMA transfer source address.

uint32_t destAddr

DMA destination address.

bool enableSrcIncrement

Source address increase after each transfer.

dma_transfer_size_t srcSize

Source transfer size unit.

bool enableDestIncrement

Destination address increase after each transfer.

dma_transfer_size_t destSize

Destination transfer unit.

uint32_t transferSize

The number of bytes to be transferred.

```
struct_dma_channel_link_config
```

#include <fsl_dma.h> DMA transfer configuration structure.

Public Members

dma_channel_link_type_t linkType

Channel link type.

uint32_t channel1

The index of channel 1.

uint32_t channel2

The index of channel 2.

```
struct_dma_handle
```

#include <fsl_dma.h> DMA DMA handle structure.

Public Members

DMA_Type *base

DMA peripheral address.

uint8_t channel

DMA channel used.

dma_callback callback

DMA callback function.

void *userData

Callback parameter.

2.7 DMAMUX: Direct Memory Access Multiplexer Driver

void DMAMUX_Init(DMAMUX_Type *base)

Initializes the DMAMUX peripheral.

This function ungates the DMAMUX clock.

Parameters

- base – DMAMUX peripheral base address.

void DMAMUX_Deinit(DMAMUX_Type *base)

Deinitializes the DMAMUX peripheral.

This function gates the DMAMUX clock.

Parameters

- base – DMAMUX peripheral base address.

static inline void DMAMUX_EnableChannel(DMAMUX_Type *base, uint32_t channel)

Enables the DMAMUX channel.

This function enables the DMAMUX channel.

Parameters

- base – DMAMUX peripheral base address.
- channel – DMAMUX channel number.

static inline void DMAMUX_DisableChannel(DMAMUX_Type *base, uint32_t channel)

Disables the DMAMUX channel.

This function disables the DMAMUX channel.

Note: The user must disable the DMAMUX channel before configuring it.

Parameters

- base – DMAMUX peripheral base address.
- channel – DMAMUX channel number.

static inline void DMAMUX_SetSource(DMAMUX_Type *base, uint32_t channel, int32_t source)

Configures the DMAMUX channel source.

Parameters

- base – DMAMUX peripheral base address.
- channel – DMAMUX channel number.
- source – Channel source, which is used to trigger the DMA transfer. User need to use the dma_request_source_t type as the input parameter.

static inline void DMAMUX_EnablePeriodTrigger(DMAMUX_Type *base, uint32_t channel)

Enables the DMAMUX period trigger.

This function enables the DMAMUX period trigger feature.

Parameters

- base – DMAMUX peripheral base address.
- channel – DMAMUX channel number.

static inline void DMAMUX_DisablePeriodTrigger(DMAMUX_Type *base, uint32_t channel)

Disables the DMAMUX period trigger.

This function disables the DMAMUX period trigger.

Parameters

- base – DMAMUX peripheral base address.
- channel – DMAMUX channel number.

FSL_DMAMUX_DRIVER_VERSION

DMAMUX driver version 2.1.1.

2.8 EWM: External Watchdog Monitor Driver

void EWM_Init(EWM_Type *base, const *ewm_config_t* *config)

Initializes the EWM peripheral.

This function is used to initialize the EWM. After calling, the EWM runs immediately according to the configuration. Note that, except for the interrupt enable control bit, other control bits and registers are write once after a CPU reset. Modifying them more than once generates a bus transfer error.

This is an example.

```
ewm_config_t config;
EWM_GetDefaultConfig(&config);
config.compareHighValue = 0xAAU;
EWM_Init(ewm_base,&config);
```

Parameters

- base – EWM peripheral base address
- config – The configuration of the EWM

void EWM_Deinit(EWM_Type *base)

Deinitializes the EWM peripheral.

This function is used to shut down the EWM.

Parameters

- base – EWM peripheral base address

void EWM_GetDefaultConfig(*ewm_config_t* *config)

Initializes the EWM configuration structure.

This function initializes the EWM configuration structure to default values. The default values are as follows.

```
ewmConfig->enableEwm = true;
ewmConfig->enableEwmInput = false;
ewmConfig->setInputAssertLogic = false;
ewmConfig->enableInterrupt = false;
ewmConfig->ewm_lpo_clock_source_t = kEWM_LpoClockSource0;
ewmConfig->prescaler = 0;
ewmConfig->compareLowValue = 0;
ewmConfig->compareHighValue = 0xFEU;
```

See also:

ewm_config_t

Parameters

- config – Pointer to the EWM configuration structure.

static inline void EWM_EnableInterrupts(EWM_Type *base, uint32_t mask)

Enables the EWM interrupt.

This function enables the EWM interrupt.

Parameters

- base – EWM peripheral base address
- mask – The interrupts to enable The parameter can be combination of the following source if defined
 - kEWM_InterruptEnable

static inline void EWM_DisableInterrupts(EWM_Type *base, uint32_t mask)

Disables the EWM interrupt.

This function enables the EWM interrupt.

Parameters

- base – EWM peripheral base address
- mask – The interrupts to disable The parameter can be combination of the following source if defined
 - kEWM_InterruptEnable

static inline uint32_t EWM_GetStatusFlags(EWM_Type *base)

Gets all status flags.

This function gets all status flags.

This is an example for getting the running flag.

```
uint32_t status;  
status = EWM_GetStatusFlags(ewm_base) & kEWM_RunningFlag;
```

See also:

_ewm_status_flags_t

- True: a related status flag has been set.
- False: a related status flag is not set.

Parameters

- base – EWM peripheral base address

Returns

State of the status flag: asserted (true) or not-asserted (false).

void EWM_Refresh(EWM_Type *base)

Services the EWM.

This function resets the EWM counter to zero.

Parameters

- base – EWM peripheral base address

FSL_EWM_DRIVER_VERSION

EWM driver version 2.0.4.

enum _ewm_lpo_clock_source

Describes EWM clock source.

Values:

enumerator kEWM_LpoClockSource0

EWM clock sourced from lpo_clk[0]

enumerator kEWM_LpoClockSource1

EWM clock sourced from lpo_clk[1]

enumerator kEWM_LpoClockSource2

EWM clock sourced from lpo_clk[2]

enumerator kEWM_LpoClockSource3

EWM clock sourced from lpo_clk[3]

enum _ewm_interrupt_enable_t

EWM interrupt configuration structure with default settings all disabled.

This structure contains the settings for all of EWM interrupt configurations.

Values:

enumerator kEWM_InterruptEnable

Enable the EWM to generate an interrupt

enum _ewm_status_flags_t

EWM status flags.

This structure contains the constants for the EWM status flags for use in the EWM functions.

Values:

enumerator kEWM_RunningFlag

Running flag, set when EWM is enabled

typedef enum _ewm_lpo_clock_source ewm_lpo_clock_source_t

Describes EWM clock source.

typedef struct _ewm_config ewm_config_t

Data structure for EWM configuration.

This structure is used to configure the EWM.

struct _ewm_config

#include <fsl_ewm.h> Data structure for EWM configuration.

This structure is used to configure the EWM.

Public Members

bool enableEwm

Enable EWM module

bool enableEwmInput

Enable EWM_in input

bool setInputAssertLogic

EWM_in signal assertion state

`bool enableInterrupt`
Enable EWM interrupt

`ewm_lpo_clock_source_t clockSource`
Clock source select

`uint8_t prescaler`
Clock prescaler value

`uint8_t compareLowValue`
Compare low-register value

`uint8_t compareHighValue`
Compare high-register value

2.9 FGPIO Driver

`void FGPIO_PinInit(FGPIO_Type *base, uint32_t pin, const gpio_pin_config_t *config)`

Initializes a FGPIO pin used by the board.

To initialize the FGPIO driver, define a pin configuration, as either input or output, in the user file. Then, call the `FGPIO_PinInit()` function.

This is an example to define an input pin or an output pin configuration:

```
Define a digital input pin configuration,
gpio_pin_config_t config =
{
    kGPIO_DigitalInput,
    0,
}
Define a digital output pin configuration,
gpio_pin_config_t config =
{
    kGPIO_DigitalOutput,
    0,
}
```

Parameters

- `base` – FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
- `pin` – FGPIO port pin number
- `config` – FGPIO pin configuration pointer

`static inline void FGPIO_PinWrite(FGPIO_Type *base, uint32_t pin, uint8_t output)`

Sets the output level of the multiple FGPIO pins to the logic 1 or 0.

Parameters

- `base` – FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
- `pin` – FGPIO pin number
- `output` – FGPIOpin output logic level.
 - 0: corresponding pin output low-logic level.
 - 1: corresponding pin output high-logic level.


```
static inline void FGPIO_PortSet(FGPIO_Type *base, uint32_t mask)
```

Sets the output level of the multiple FGPIO pins to the logic 1.

Parameters

- base – FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
- mask – FGPIO pin number macro

```
static inline void FGPIO_PortClear(FGPIO_Type *base, uint32_t mask)
```

Sets the output level of the multiple FGPIO pins to the logic 0.

Parameters

- base – FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
- mask – FGPIO pin number macro

```
static inline void FGPIO_PortToggle(FGPIO_Type *base, uint32_t mask)
```

Reverses the current output logic of the multiple FGPIO pins.

Parameters

- base – FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
- mask – FGPIO pin number macro

```
static inline uint32_t FGPIO_PinRead(FGPIO_Type *base, uint32_t pin)
```

Reads the current input value of the FGPIO port.

Parameters

- base – FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
- pin – FGPIO pin number

Return values

FGPIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

```
uint32_t FGPIO_PortGetInterruptFlags(FGPIO_Type *base)
```

Reads the FGPIO port interrupt status flag.

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level-sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

- base – FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)

Return values

The – current FGPIO port interrupt status flags, for example, 0x00010001 means the pin 0 and 17 have the interrupt.

```
void FGPIO_PortClearInterruptFlags(FGPIO_Type *base, uint32_t mask)
```

Clears the multiple FGPIO pin interrupt status flag.

Parameters

- base – FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
- mask – FGPIO pin number macro

void FGPIO_CheckAttributeBytes(FGPIO_Type *base, *gpio_checker_attribute_t* attribute)

The FGPIO module supports a device-specific number of data ports, organized as 32-bit words. Each 32-bit data port includes a GACR register, which defines the byte-level attributes required for a successful access to the GPIO programming model. The attribute controls for the 4 data bytes in the GACR follow a standard little endian data convention.

Parameters

- base – FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
- attribute – FGPIO checker attribute

2.10 C90TFS Flash Driver

2.11 ftfx adapter

2.12 Ftfx CACHE Driver

enum *_ftfx_cache_ram_func_constants*

Constants for execute-in-RAM flash function.

Values:

enumerator kFTFx_CACHE_RamFuncMaxSizeInWords

The maximum size of execute-in-RAM function.

typedef struct *_flash_prefetch_speculation_status* ftfx_prefetch_speculation_status_t

FTFx prefetch speculation status.

typedef struct *_ftfx_cache_config* ftfx_cache_config_t

FTFx cache driver state information.

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

status_t FTFx_CACHE_Init(*ftfx_cache_config_t* *config)

Initializes the global FTFx cache structure members.

This function checks and initializes the Flash module for the other FTFx cache APIs.

Parameters

- config – Pointer to the storage for the driver runtime state.

Return values

- kStatus_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

```
status_t FTFx_CACHE_ClearCachePrefetchSpeculation(ftfx_cache_config_t *config, bool
                                                    isPreProcess)
```

Process the cache/prefetch/speculation to the flash.

Parameters

- config – A pointer to the storage for the driver runtime state.
- isPreProcess – The possible option used to control flash cache/prefetch/speculation

Return values

- kStatus_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidArgument – Invalid argument is provided.
- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

```
status_t FTFx_CACHE_PflashSetPrefetchSpeculation(ftfx_prefetch_speculation_status_t
                                                    *speculationStatus)
```

Sets the PFlash prefetch speculation to the intended speculation status.

Parameters

- speculationStatus – The expected protect status to set to the PFlash protection register. Each bit is

Return values

- kStatus_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidSpeculationOption – An invalid speculation option argument is provided.

```
status_t FTFx_CACHE_PflashGetPrefetchSpeculation(ftfx_prefetch_speculation_status_t
                                                    *speculationStatus)
```

Gets the PFlash prefetch speculation status.

Parameters

- speculationStatus – Speculation status returned by the PFlash IP.

Return values

kStatus_FTFx_Success – API was executed successfully.

```
struct _flash_prefetch_speculation_status
#include <fsl_ftfx_cache.h> FTFx prefetch speculation status.
```

Public Members

```
bool instructionOff
    Instruction speculation.
```

```
bool dataOff
    Data speculation.
```

```
union function_bit_operation_ptr_t
#include <fsl_ftfx_cache.h>
```

Public Members

```
uint32_t commadAddr
```

```
void (*callFlashCommand)(volatile uint32_t *base, uint32_t bitMask, uint32_t bitShift,  
uint32_t bitValue)
```

```
struct _ftfx_cache_config
```

#include <fsl_ftfx_cache.h> FTFx cache driver state information.

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

Public Members

```
uint8_t flashMemoryIndex
```

0 - primary flash; 1 - secondary flash

```
function_bit_operation_ptr_t bitOperFuncAddr
```

An buffer point to the flash execute-in-RAM function.

2.13 ftfx controller

FTFx driver status codes.

Values:

```
enumerator kStatus_FTFx_Success
```

API is executed successfully

```
enumerator kStatus_FTFx_InvalidArgument
```

Invalid argument

```
enumerator kStatus_FTFx_SizeError
```

Error size

```
enumerator kStatus_FTFx_AlignmentError
```

Parameter is not aligned with the specified baseline

```
enumerator kStatus_FTFx_AddressError
```

Address is out of range

```
enumerator kStatus_FTFx_AccessError
```

Invalid instruction codes and out-of bound addresses

```
enumerator kStatus_FTFx_ProtectionViolation
```

The program/erase operation is requested to execute on protected areas

```
enumerator kStatus_FTFx_CommandFailure
```

Run-time error during command execution.

```
enumerator kStatus_FTFx_UnknownProperty
```

Unknown property.

```
enumerator kStatus_FTFx_EraseKeyError
```

API erase key is invalid.

```
enumerator kStatus_FTFx_RegionExecuteOnly
```

The current region is execute-only.

```
enumerator kStatus_FTFx_ExecuteInRamFunctionNotReady
```

Execute-in-RAM function is not available.

enumerator kStatus_FTFx_PartitionStatusUpdateFailure
Failed to update partition status.

enumerator kStatus_FTFx_SetFlexramAsEepromError
Failed to set FlexRAM as EEPROM.

enumerator kStatus_FTFx_RecoverFlexramAsRamError
Failed to recover FlexRAM as RAM.

enumerator kStatus_FTFx_SetFlexramAsRamError
Failed to set FlexRAM as RAM.

enumerator kStatus_FTFx_RecoverFlexramAsEepromError
Failed to recover FlexRAM as EEPROM.

enumerator kStatus_FTFx_CommandNotSupported
Flash API is not supported.

enumerator kStatus_FTFx_SwapSystemNotInUninitialized
Swap system is not in an uninitialized state.

enumerator kStatus_FTFx_SwapIndicatorAddressError
The swap indicator address is invalid.

enumerator kStatus_FTFx_ReadOnlyProperty
The flash property is read-only.

enumerator kStatus_FTFx_InvalidPropertyValue
The flash property value is out of range.

enumerator kStatus_FTFx_InvalidSpeculationOption
The option of flash prefetch speculation is invalid.

enumerator kStatus_FTFx_CommandOperationInProgress
The option of flash command is processing.

enum _ftfx_driver_api_keys
Enumeration for FTFx driver API keys.

Note: The resulting value is built with a byte order such that the string being readable in expected order when viewed in a hex editor, if the value is treated as a 32-bit little endian value.

Values:

enumerator kFTFx_ApiEraseKey
Key value used to validate all FTFx erase APIs.

void FTFx_API_Init(*ftfx_config_t* *config)
Initializes the global flash properties structure members.
This function checks and initializes the Flash module for the other Flash APIs.

Parameters

- config – Pointer to the storage for the driver runtime state.

status_t FTFx_API_UpdateFlexnvmPartitionStatus(*ftfx_config_t* *config)
Updates FlexNVM memory partition status according to data flash 0 IFR.
This function updates FlexNVM memory partition status.

Parameters

- `config` – Pointer to the storage for the driver runtime state.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_PartitionStatusUpdateFailure` – Failed to update the partition status.

`status_t FTFx_CMD_Erase(ftfx_config_t *config, uint32_t start, uint32_t lengthInBytes, uint32_t key)`

Erases the flash sectors encompassed by parameters passed into function.

This function erases the appropriate number of flash sectors based on the desired start address and length.

Parameters

- `config` – The pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be erased. The start address does not need to be sector-aligned but must be word-aligned.
- `lengthInBytes` – The length, given in bytes (not words or long-words) to be erased. Must be word-aligned.
- `key` – The value used to validate all flash erase APIs.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – The parameter is not aligned with the specified baseline.
- `kStatus_FTFx_AddressError` – The address is out of range.
- `kStatus_FTFx_EraseKeyError` – The API erase key is invalid.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FTFx_CMD_EraseSectorNonBlocking(ftfx_config_t *config, uint32_t start, uint32_t key)`

Erases the flash sectors encompassed by parameters passed into function.

This function erases one flash sector size based on the start address.

Parameters

- `config` – The pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be erased. The start address does not need to be sector-aligned but must be word-aligned.
- `key` – The value used to validate all flash erase APIs.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.

- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – The parameter is not aligned with the specified baseline.
- `kStatus_FTFx_AddressError` – The address is out of range.
- `kStatus_FTFx_EraseKeyError` – The API erase key is invalid.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.

`status_t FTFx_CMD_EraseAll(ftfx_config_t *config, uint32_t key)`

Erases entire flash.

Parameters

- `config` – Pointer to the storage for the driver runtime state.
- `key` – A value used to validate all flash erase APIs.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_EraseKeyError` – API erase key is invalid.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.
- `kStatus_FTFx_PartitionStatusUpdateFailure` – Failed to update the partition status.

`status_t FTFx_CMD_EraseAllUnsecure(ftfx_config_t *config, uint32_t key)`

Erases the entire flash, including protected sectors.

Parameters

- `config` – Pointer to the storage for the driver runtime state.
- `key` – A value used to validate all flash erase APIs.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_EraseKeyError` – API erase key is invalid.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.

- `kStatus_FTFx_PartitionStatusUpdateFailure` – Failed to update the partition status.

`status_t FTFx_CMD_EraseAllExecuteOnlySegments(ftfx_config_t *config, uint32_t key)`

Erases all program flash execute-only segments defined by the FXACC registers.

Parameters

- `config` – Pointer to the storage for the driver runtime state.
- `key` – A value used to validate all flash erase APIs.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_EraseKeyError` – API erase key is invalid.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FTFx_CMD_Program(ftfx_config_t *config, uint32_t start, const uint8_t *src, uint32_t lengthInBytes)`

Programs flash with data at locations passed in through parameters.

This function programs the flash memory with the desired data for a given flash area as determined by the start address and the length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be programmed. Must be word-aligned.
- `src` – A pointer to the source buffer of data that is to be programmed into the flash.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with the specified baseline.
- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.

- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FTFx_CMD_ProgramOnce(ftfx_config_t *config, uint32_t index, const uint8_t *src, uint32_t lengthInBytes)`

Programs Program Once Field through parameters.

This function programs the Program Once Field with the desired data for a given flash area as determined by the index and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `index` – The index indicating which area of the Program Once Field to be programmed.
- `src` – A pointer to the source buffer of data that is to be programmed into the Program Once Field.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FTFx_CMD_ProgramSection(ftfx_config_t *config, uint32_t start, const uint8_t *src, uint32_t lengthInBytes)`

Programs flash with data at locations passed in through parameters via the Program Section command.

This function programs the flash memory with the desired data for a given flash area as determined by the start address and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be programmed. Must be word-aligned.
- `src` – A pointer to the source buffer of data that is to be programmed into the flash.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with specified baseline.

- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FTFx_SetFlexramAsRamError` – Failed to set flexram as RAM.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.
- `kStatus_FTFx_RecoverFlexramAsEepromError` – Failed to recover FlexRAM as EEPROM.

`status_t FTFx_CMD_ProgramPartition(ftfx_config_t *config, ftfx_partition_flexram_load_opt_t option, uint32_t eepromDataSizeCode, uint32_t flexnvmPartitionCode, uint8_t CSEcKeySize, uint8_t CFE)`

Prepares the FlexNVM block for use as data flash, EEPROM backup, or a combination of both and initializes the FlexRAM.

Parameters

- `config` – Pointer to storage for the driver runtime state.
- `option` – The option used to set FlexRAM load behavior during reset.
- `eepromDataSizeCode` – Determines the amount of FlexRAM used in each of the available EEPROM subsystems.
- `flexnvmPartitionCode` – Specifies how to split the FlexNVM block between data flash memory and EEPROM backup memory supporting EEPROM functions.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – Invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.

`status_t FTFx_CMD_ReadOnce(ftfx_config_t *config, uint32_t index, uint8_t *dst, uint32_t lengthInBytes)`

Reads the Program Once Field through parameters.

This function reads the read once field with given index and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `index` – The index indicating the area of program once field to be read.
- `dst` – A pointer to the destination buffer of data that is used to store data to be read.

- `lengthInBytes` – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FTFx_CMD_ReadResource(ftfx_config_t *config, uint32_t start, uint8_t *dst, uint32_t lengthInBytes, ftfx_read_resource_opt_t option)`

Reads the resource with data at locations passed in through parameters.

This function reads the flash memory with the desired location for a given flash area as determined by the start address and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be programmed. Must be word-aligned.
- `dst` – A pointer to the destination buffer of data that is used to store data to be read.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be read. Must be word-aligned.
- `option` – The resource option which indicates which area should be read back.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with the specified baseline.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FTFx_CMD_VerifyErase(ftfx_config_t *config, uint32_t start, uint32_t lengthInBytes, ftfx_margin_value_t margin)`

Verifies an erasure of the desired flash area at a specified margin level.

This function checks the appropriate number of flash sectors based on the desired start address and length to check whether the flash is erased to the specified read margin level.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be verified. The start address does not need to be sector-aligned but must be word-aligned.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
- `margin` – Read margin choice.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FTFx_CMD_VerifyEraseAll(ftfx_config_t *config, ftfx_margin_value_t margin)`

Verifies erasure of the entire flash at a specified margin level.

This function checks whether the flash is erased to the specified read margin level.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `margin` – Read margin choice.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FTFx_CMD_VerifyEraseAllExecuteOnlySegments(ftfx_config_t *config,
ftfx_margin_value_t margin)`

Verifies whether the program flash execute-only segments have been erased to the specified read margin level.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `margin` – Read margin choice.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FTFx_CMD_VerifyProgram(ftfx_config_t *config, uint32_t start, uint32_t lengthInBytes, const uint8_t *expectedData, ftfx_margin_value_t margin, uint32_t *failedAddress, uint32_t *failedData)`

Verifies programming of the desired flash area at a specified margin level.

This function verifies the data programed in the flash memory using the Flash Program Check Command and compares it to the expected data for a given flash area as determined by the start address and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be verified. Must be word-aligned.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
- `expectedData` – A pointer to the expected data that is to be verified against.
- `margin` – Read margin choice.
- `failedAddress` – A pointer to the returned failing address.
- `failedData` – A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.

- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FTFx_REG_GetSecurityState(ftfx_config_t *config, ftfx_security_state_t *state)`

Returns the security state via the pointer passed into the function.

This function retrieves the current flash security status, including the security enabling state and the backdoor key enabling state.

Parameters

- `config` – A pointer to storage for the driver runtime state.
- `state` – A pointer to the value returned for the current security status code:

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.

`status_t FTFx_CMD_SecurityBypass(ftfx_config_t *config, const uint8_t *backdoorKey)`

Allows users to bypass security with a backdoor key.

If the MCU is in secured state, this function unsecures the MCU by comparing the provided backdoor key with ones in the flash configuration field.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `backdoorKey` – A pointer to the user buffer containing the backdoor key.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FTFx_CMD_SetFlexramFunction(ftfx_config_t *config, ftfx_flexram_func_opt_t option)`

Sets the FlexRAM function command.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `option` – The option used to set the work mode of FlexRAM.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.

- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

```
status_t FTFx_CMD_SwapControl(ftfx_config_t *config, uint32_t address,
                             ftfx_swap_control_opt_t option, ftfx_swap_state_config_t
                             *returnInfo)
```

Configures the Swap function or checks the swap state of the Flash module.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `address` – Address used to configure the flash Swap function.
- `option` – The possible option used to configure Flash Swap function or check the flash Swap status
- `returnInfo` – A pointer to the data which is used to return the information of flash Swap.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FTFx_SwapIndicatorAddressError` – Swap indicator address is invalid.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

```
enum _ftfx_partition_flexram_load_option
```

Enumeration for the FlexRAM load during reset option.

Values:

```
enumerator kFTFx_PartitionFlexramLoadOptLoadedWithValidEepromData
    FlexRAM is loaded with valid EEPROM data during reset sequence.
```

```
enumerator kFTFx_PartitionFlexramLoadOptNotLoaded
    FlexRAM is not loaded during reset sequence.
```

```
enum _ftfx_read_resource_opt
```

Enumeration for the two possible options of flash read resource command.

Values:

```
enumerator kFTFx_ResourceOptionFlashIfR
    Select code for Program flash 0 IFR, Program flash swap 0 IFR, Data flash 0 IFR
```

enumerator kFTFx_ResourceOptionVersionId

Select code for the version ID

enum _ftfx_margin_value

Enumeration for supported FTFx margin levels.

Values:

enumerator kFTFx_MarginValueNormal

Use the 'normal' read level for 1s.

enumerator kFTFx_MarginValueUser

Apply the 'User' margin to the normal read-1 level.

enumerator kFTFx_MarginValueFactory

Apply the 'Factory' margin to the normal read-1 level.

enumerator kFTFx_MarginValueInvalid

Not real margin level, Used to determine the range of valid margin level.

enum _ftfx_security_state

Enumeration for the three possible FTFx security states.

Values:

enumerator kFTFx_SecurityStateNotSecure

Flash is not secure.

enumerator kFTFx_SecurityStateBackdoorEnabled

Flash backdoor is enabled.

enumerator kFTFx_SecurityStateBackdoorDisabled

Flash backdoor is disabled.

enum _ftfx_flexram_function_option

Enumeration for the two possible options of set FlexRAM function command.

Values:

enumerator kFTFx_FlexramFuncOptAvailableAsRam

An option used to make FlexRAM available as RAM

enumerator kFTFx_FlexramFuncOptEepromQuickWriteRecovery

An option used to complete interrupted EEPROM quick write process

enumerator kFTFx_FlexramFuncOptEepromQuickWriteStatus

An option used to make EEPROM quick write status query

enumerator kFTFx_FlexramFuncOptAvailableForEepromQuickWrite

An option used to make FlexRAM available for EEPROM in Quick Write mode

enumerator kFTFx_FlexramFuncOptAvailableForEeprom

An option used to make FlexRAM available for EEPROM

enum _flash_acceleration_ram_property

Enumeration for acceleration ram property.

Values:

enumerator kFLASH_AccelerationRamSize

enum _ftfx_swap_control_option

Enumeration for the possible options of Swap control commands.

Values:

enumerator kFTFx_SwapControlOptionIntializeSystem

An option used to initialize the Swap system

enumerator kFTFx_SwapControlOptionSetInUpdateState

An option used to set the Swap in an update state

enumerator kFTFx_SwapControlOptionSetInCompleteState

An option used to set the Swap in a complete state

enumerator kFTFx_SwapControlOptionReportStatus

An option used to report the Swap status

enumerator kFTFx_SwapControlOptionDisableSystem

An option used to disable the Swap status

enum _ftfx_swap_state

Enumeration for the possible flash Swap status.

Values:

enumerator kFTFx_SwapStateUninitialized

Flash Swap system is in an uninitialized state.

enumerator kFTFx_SwapStateReady

Flash Swap system is in a ready state.

enumerator kFTFx_SwapStateUpdate

Flash Swap system is in an update state.

enumerator kFTFx_SwapStateUpdateErased

Flash Swap system is in an updateErased state.

enumerator kFTFx_SwapStateComplete

Flash Swap system is in a complete state.

enumerator kFTFx_SwapStateDisabled

Flash Swap system is in a disabled state.

enum _ftfx_swap_block_status

Enumeration for the possible flash Swap block status.

Values:

enumerator kFTFx_SwapBlockStatusLowerHalfProgramBlocksAtZero

Swap block status is that lower half program block at zero.

enumerator kFTFx_SwapBlockStatusUpperHalfProgramBlocksAtZero

Swap block status is that upper half program block at zero.

enum _ftfx_memory_type

Enumeration for FTFx memory type.

Values:

enumerator kFTFx_MemTypePflash

enumerator kFTFx_MemTypeFlexnvm

typedef enum _ftfx_partition_flexram_load_option ftfx_partition_flexram_load_opt_t

Enumeration for the FlexRAM load during reset option.

typedef enum _ftfx_read_resource_opt ftfx_read_resource_opt_t

Enumeration for the two possible options of flash read resource command.

typedef enum *_ftfx_margin_value* ftfx_margin_value_t

Enumeration for supported FTFx margin levels.

typedef enum *_ftfx_security_state* ftfx_security_state_t

Enumeration for the three possible FTFx security states.

typedef enum *_ftfx_flexram_function_option* ftfx_flexram_func_opt_t

Enumeration for the two possible options of set FlexRAM function command.

typedef enum *_ftfx_swap_control_option* ftfx_swap_control_opt_t

Enumeration for the possible options of Swap control commands.

typedef enum *_ftfx_swap_state* ftfx_swap_state_t

Enumeration for the possible flash Swap status.

typedef enum *_ftfx_swap_block_status* ftfx_swap_block_status_t

Enumeration for the possible flash Swap block status.

typedef struct *_ftfx_swap_state_config* ftfx_swap_state_config_t

Flash Swap information.

typedef struct *_ftfx_special_mem* ftfx_spec_mem_t

ftfx special memory access information.

typedef struct *_ftfx_mem_descriptor* ftfx_mem_desc_t

Flash memory descriptor.

typedef struct *_ftfx_ops_config* ftfx_ops_config_t

Active FTFx information for the current operation.

typedef struct *_ftfx_ifr_descriptor* ftfx_ifr_desc_t

Flash IFR memory descriptor.

typedef struct *_ftfx_config* ftfx_config_t

Flash driver state information.

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

struct *_ftfx_swap_state_config*

#include <fsl_ftfx_controller.h> Flash Swap information.

Public Members

ftfx_swap_state_t flashSwapState

The current Swap system status.

ftfx_swap_block_status_t currentSwapBlockStatus

The current Swap block status.

ftfx_swap_block_status_t nextSwapBlockStatus

The next Swap block status.

struct *_ftfx_special_mem*

#include <fsl_ftfx_controller.h> ftfx special memory access information.

Public Members

uint32_t base

Base address of flash special memory.

uint32_t size
size of flash special memory.

uint32_t count
flash special memory count.

struct _ftfx_mem_descriptor
#include <fsl_ftfx_controller.h> Flash memory descriptor.

Public Members

uint32_t blockBase
A base address of the flash block

uint32_t aliasBlockBase
A base address of the alias flash block

uint32_t totalSize
The size of the flash block.

uint32_t sectorSize
The size in bytes of a sector of flash.

uint32_t blockCount
A number of flash blocks.

struct _ftfx_ops_config
#include <fsl_ftfx_controller.h> Active FTFx information for the current operation.

Public Members

uint32_t convertedAddress
A converted address for the current flash type.

struct _ftfx_ifr_descriptor
#include <fsl_ftfx_controller.h> Flash IFR memory descriptor.

union function_ptr_t
#include <fsl_ftfx_controller.h>

Public Members

uint32_t commadAddr

void (*callFlashCommand)(volatile uint8_t *FTMRx_fstat)

struct _ftfx_config
#include <fsl_ftfx_controller.h> Flash driver state information.

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

Public Members

uint32_t flexramBlockBase
The base address of the FlexRAM/acceleration RAM

uint32_t flexramTotalSize

The size of the FlexRAM/acceleration RAM

uint16_t eepromTotalSize

The size of EEPROM area which was partitioned from FlexRAM

function_ptr_t runCmdFuncAddr

An buffer point to the flash execute-in-RAM function.

struct ____unnamed12____

Public Members

uint8_t type

Type of flash block.

uint8_t index

Index of flash block.

struct feature

struct addrAligment

struct feature

struct resRange

Public Members

uint8_t versionIdStart

Version ID start address

uint32_t pflashIfrStart

Program Flash 0 IFR start address

uint32_t dflashIfrStart

Data Flash 0 IFR start address

uint32_t pflashSwapIfrStart

Program Flash Swap IFR start address

struct idxInfo

2.14 ftfx feature

FTFx_DRIVER_IS_FLASH_RESIDENT

Flash driver location.

Used for the flash resident application.

FTFx_DRIVER_IS_EXPORTED

Flash Driver Export option.

Used for the MCUXpresso SDK application.

FTFx_FLASH1_HAS_PROT_CONTROL

Indicates whether the secondary flash has its own protection register in flash module.

FTFx_FLASH1_HAS_XACC_CONTROL

Indicates whether the secondary flash has its own Execute-Only access register in flash module.

FTFx_DRIVER_HAS_FLASH1_SUPPORT

Indicates whether the secondary flash is supported in the Flash driver.

FTFx_FLASH_COUNT

FTFx_FLASH1_IS_INDEPENDENT_BLOCK

2.15 Fftfx FLASH Driver

status_t FLASH_Init(*flash_config_t* *config)

Initializes the global flash properties structure members.

This function checks and initializes the Flash module for the other Flash APIs.

Parameters

- config – Pointer to the storage for the driver runtime state.

Return values

- kStatus_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.
- kStatus_FTFx_PartitionStatusUpdateFailure – Failed to update the partition status.

status_t FLASH_Erase(*flash_config_t* *config, uint32_t start, uint32_t lengthInBytes, uint32_t key)

Erases the Dflash sectors encompassed by parameters passed into function.

This function erases the appropriate number of flash sectors based on the desired start address and length.

Parameters

- config – The pointer to the storage for the driver runtime state.
- start – The start address of the desired flash memory to be erased. The start address does not need to be sector-aligned but must be word-aligned.
- lengthInBytes – The length, given in bytes (not words or long-words) to be erased. Must be word-aligned.
- key – The value used to validate all flash erase APIs.

Return values

- kStatus_FTFx_Success – API was executed successfully; the appropriate number of flash sectors based on the desired start address and length were erased successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_AlignmentError – The parameter is not aligned with the specified baseline.
- kStatus_FTFx_AddressError – The address is out of range.
- kStatus_FTFx_EraseKeyError – The API erase key is invalid.

- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FLASH_EraseSectorNonBlocking(flash_config_t *config, uint32_t start, uint32_t key)`

Erases the Dflash sectors encompassed by parameters passed into function.

This function erases one flash sector size based on the start address, and it is executed asynchronously.

NOTE: This function can only erase one flash sector at a time, and the other commands can be executed after the previous command has been completed.

Parameters

- `config` – The pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be erased. The start address does not need to be sector-aligned but must be word-aligned.
- `key` – The value used to validate all flash erase APIs.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – The parameter is not aligned with the specified baseline.
- `kStatus_FTFx_AddressError` – The address is out of range.
- `kStatus_FTFx_EraseKeyError` – The API erase key is invalid.

`status_t FLASH_EraseAll(flash_config_t *config, uint32_t key)`

Erases entire flexnvm.

Parameters

- `config` – Pointer to the storage for the driver runtime state.
- `key` – A value used to validate all flash erase APIs.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the all pflash and flexnvm were erased successfully, the swap and eeprom have been reset to unconfigured state.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_EraseKeyError` – API erase key is invalid.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.

- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.
- `kStatus_FTFx_PartitionStatusUpdateFailure` – Failed to update the partition status.

`status_t FLASH_EraseAllUnsecure(flash_config_t *config, uint32_t key)`

Erases the entire flexnvm, including protected sectors.

Parameters

- `config` – Pointer to the storage for the driver runtime state.
- `key` – A value used to validate all flash erase APIs.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the protected sectors of flash were reset to unprotected status.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_EraseKeyError` – API erase key is invalid.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.
- `kStatus_FTFx_PartitionStatusUpdateFailure` – Failed to update the partition status.

`status_t FLASH_Program(flash_config_t *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)`

Programs flash with data at locations passed in through parameters.

This function programs the flash memory with the desired data for a given flash area as determined by the start address and the length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be programmed. Must be word-aligned.
- `src` – A pointer to the source buffer of data that is to be programmed into the flash.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the desired data were programmed successfully into flash based on desired start address and length.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with the specified baseline.
- `kStatus_FTFx_AddressError` – Address is out of range.

- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t` FLASH_ProgramOnce(*flash_config_t* *config, uint32_t index, uint8_t *src, uint32_t lengthInBytes)

Program the Program-Once-Field through parameters.

This function Program the Program-once-field with given index and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `index` – The index indicating the area of program once field to be read.
- `src` – A pointer to the source buffer of data that is used to store data to be write.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; The index indicating the area of program once field was programmed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t` FLASH_ProgramSection(*flash_config_t* *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)

Programs flash with data at locations passed in through parameters via the Program Section command.

This function programs the flash memory with the desired data for a given flash area as determined by the start address and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be programmed. Must be word-aligned.
- `src` – A pointer to the source buffer of data that is to be programmed into the flash.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the desired data have been programmed successfully into flash based on start address and length.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FTFx_SetFlexramAsRamError` – Failed to set flexram as RAM.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.
- `kStatus_FTFx_RecoverFlexramAsEepromError` – Failed to recover FlexRAM as EEPROM.

`status_t FLASH_ReadResource(flash_config_t *config, uint32_t start, uint8_t *dst, uint32_t lengthInBytes, ftfx_read_resource_opt_t option)`

Reads the resource with data at locations passed in through parameters.

This function reads the flash memory with the desired location for a given flash area as determined by the start address and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be programmed. Must be word-aligned.
- `dst` – A pointer to the destination buffer of data that is used to store data to be read.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be read. Must be word-aligned.
- `option` – The resource option which indicates which area should be read back.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the data have been read successfully from program flash IFR, data flash IFR space, and the Version ID field.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with the specified baseline.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.

- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FLASH_ReadOnce(flash_config_t *config, uint32_t index, uint8_t *dst, uint32_t lengthInBytes)`

Reads the Program Once Field through parameters.

This function reads the read once field with given index and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `index` – The index indicating the area of program once field to be read.
- `dst` – A pointer to the destination buffer of data that is used to store data to be read.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the data have been successfully read from Program flash0 IFR map and Program Once field based on index and length.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FLASH_VerifyErase(flash_config_t *config, uint32_t start, uint32_t lengthInBytes, ftfx_margin_value_t margin)`

Verifies an erasure of the desired flash area at a specified margin level.

This function checks the appropriate number of flash sectors based on the desired start address and length to check whether the flash is erased to the specified read margin level.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be verified. The start address does not need to be sector-aligned but must be word-aligned.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
- `margin` – Read margin choice.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the specified FLASH region has been erased.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.

- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FLASH_VerifyEraseAll(flash_config_t *config, ftfx_margin_value_t margin)`

Verifies erasure of the entire flash at a specified margin level.

This function checks whether the flash is erased to the specified read margin level.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `margin` – Read margin choice.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; all program flash and flexnvm were in erased state.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FLASH_VerifyProgram(flash_config_t *config, uint32_t start, uint32_t lengthInBytes, const uint8_t *expectedData, ftfx_margin_value_t margin, uint32_t *failedAddress, uint32_t *failedData)`

Verifies programming of the desired flash area at a specified margin level.

This function verifies the data programmed in the flash memory using the Flash Program Check Command and compares it to the expected data for a given flash area as determined by the start address and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be verified. Must be word-aligned.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
- `expectedData` – A pointer to the expected data that is to be verified against.
- `margin` – Read margin choice.
- `failedAddress` – A pointer to the returned failing address.

- `failedData` – A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the desired data have been successfully programed into specified FLASH region.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t` FLASH_GetSecurityState(*flash_config_t* *config, *ftfx_security_state_t* *state)

Returns the security state via the pointer passed into the function.

This function retrieves the current flash security status, including the security enabling state and the backdoor key enabling state.

Parameters

- `config` – A pointer to storage for the driver runtime state.
- `state` – A pointer to the value returned for the current security status code:

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the security state of flash was stored to state.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.

`status_t` FLASH_SecurityBypass(*flash_config_t* *config, `const uint8_t` *backdoorKey)

Allows users to bypass security with a backdoor key.

If the MCU is in secured state, this function unsecures the MCU by comparing the provided backdoor key with ones in the flash configuration field.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `backdoorKey` – A pointer to the user buffer containing the backdoor key.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.

- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t` FLASH_SetFlexramFunction(*flash_config_t* *config, *ftfx_flexram_func_opt_t* option)

Sets the FlexRAM function command.

Parameters

- config – A pointer to the storage for the driver runtime state.
- option – The option used to set the work mode of FlexRAM.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the FlexRAM has been successfully configured as RAM or EEPROM.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t` FLASH_Swap(*flash_config_t* *config, `uint32_t` address, `bool` isSetEnable)

Swaps the lower half flash with the higher half flash.

Parameters

- config – A pointer to the storage for the driver runtime state.
- address – Address used to configure the flash swap function
- isSetEnable – The possible option used to configure the Flash Swap function or check the flash Swap status.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the lower half flash and higher half flash have been swapped.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FTFx_SwapIndicatorAddressError` – Swap indicator address is invalid.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.

- `kStatus_FTFx_SwapSystemNotInUninitialized` – Swap system is not in an uninitialized state.

`status_t FLASH_IsProtected(flash_config_t *config, uint32_t start, uint32_t lengthInBytes, flash_prot_state_t *protection_state)`

Returns the protection state of the desired flash area via the pointer passed into the function.

This function retrieves the current flash protect status for a given flash area as determined by the start address and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be checked. Must be word-aligned.
- `lengthInBytes` – The length, given in bytes (not words or long-words) to be checked. Must be word-aligned.
- `protection_state` – A pointer to the value returned for the current protection status code for the desired flash area.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the protection state of specified FLASH region was stored to `protection_state`.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FTFx_AddressError` – The address is out of range.

`status_t FLASH_IsExecuteOnly(flash_config_t *config, uint32_t start, uint32_t lengthInBytes, flash_xacc_state_t *access_state)`

Returns the access state of the desired flash area via the pointer passed into the function.

This function retrieves the current flash access status for a given flash area as determined by the start address and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be checked. Must be word-aligned.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be checked. Must be word-aligned.
- `access_state` – A pointer to the value returned for the current access status code for the desired flash area.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the executeOnly state of specified FLASH region was stored to `access_state`.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – The parameter is not aligned to the specified baseline.
- `kStatus_FTFx_AddressError` – The address is out of range.

status_t FLASH_PflashSetProtection(*flash_config_t* *config, *pflash_prot_status_t* *protectStatus)
Sets the PFlash Protection to the intended protection status.

Parameters

- config – A pointer to storage for the driver runtime state.
- protectStatus – The expected protect status to set to the PFlash protection register. Each bit is corresponding to protection of 1/32(64) of the total PFlash. The least significant bit is corresponding to the lowest address area of PFlash. The most significant bit is corresponding to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

- kStatus_FTFx_Success – API was executed successfully; the specified FLASH region is protected.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_CommandFailure – Run-time error during command execution.

status_t FLASH_PflashGetProtection(*flash_config_t* *config, *pflash_prot_status_t* *protectStatus)
Gets the PFlash protection status.

Parameters

- config – A pointer to the storage for the driver runtime state.
- protectStatus – Protect status returned by the PFlash IP. Each bit is corresponding to the protection of 1/32(64) of the total PFlash. The least significant bit corresponds to the lowest address area of the PFlash. The most significant bit corresponds to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

- kStatus_FTFx_Success – API was executed successfully; the Protection state was stored to protectStatus;
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

status_t FLASH_GetProperty(*flash_config_t* *config, *flash_property_tag_t* whichProperty, *uint32_t* *value)

Returns the desired flash property.

Parameters

- config – A pointer to the storage for the driver runtime state.
- whichProperty – The desired property from the list of properties in enum *flash_property_tag_t*
- value – A pointer to the value returned for the desired flash property.

Return values

- kStatus_FTFx_Success – API was executed successfully; the flash property was stored to value.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_UnknownProperty – An unknown property tag.

`status_t FLASH_GetCommandState(void)`

Get previous command status.

This function is used to obtain the execution status of the previous command.

Return values

- `kStatus_FTFx_Success` – The previous command is executed successfully.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`FSL_FLASH_DRIVER_VERSION`

Flash driver version for SDK.

Version 3.3.0.

`FSL_FLASH_DRIVER_VERSION_ROM`

Flash driver version for ROM.

Version 3.0.0.

`enum _flash_protection_state`

Enumeration for the three possible flash protection levels.

Values:

enumerator `kFLASH_ProtectionStateUnprotected`

Flash region is not protected.

enumerator `kFLASH_ProtectionStateProtected`

Flash region is protected.

enumerator `kFLASH_ProtectionStateMixed`

Flash is mixed with protected and unprotected region.

`enum _flash_execute_only_access_state`

Enumeration for the three possible flash execute access levels.

Values:

enumerator `kFLASH_AccessStateUnLimited`

Flash region is unlimited.

enumerator `kFLASH_AccessStateExecuteOnly`

Flash region is execute only.

enumerator `kFLASH_AccessStateMixed`

Flash is mixed with unlimited and execute only region.

`enum _flash_property_tag`

Enumeration for various flash properties.

Values:

enumerator `kFLASH_PropertyPflash0SectorSize`

Pflash sector size property.

enumerator kFLASH_PropertyPflash0TotalSize
Pflash total size property.

enumerator kFLASH_PropertyPflash0BlockSize
Pflash block size property.

enumerator kFLASH_PropertyPflash0BlockCount
Pflash block count property.

enumerator kFLASH_PropertyPflash0BlockBaseAddr
Pflash block base address property.

enumerator kFLASH_PropertyPflash0FacSupport
Pflash fac support property.

enumerator kFLASH_PropertyPflash0AccessSegmentSize
Pflash access segment size property.

enumerator kFLASH_PropertyPflash0AccessSegmentCount
Pflash access segment count property.

enumerator kFLASH_PropertyPflash1SectorSize
Pflash sector size property.

enumerator kFLASH_PropertyPflash1TotalSize
Pflash total size property.

enumerator kFLASH_PropertyPflash1BlockSize
Pflash block size property.

enumerator kFLASH_PropertyPflash1BlockCount
Pflash block count property.

enumerator kFLASH_PropertyPflash1BlockBaseAddr
Pflash block base address property.

enumerator kFLASH_PropertyPflash1FacSupport
Pflash fac support property.

enumerator kFLASH_PropertyPflash1AccessSegmentSize
Pflash access segment size property.

enumerator kFLASH_PropertyPflash1AccessSegmentCount
Pflash access segment count property.

enumerator kFLASH_PropertyFlexRamBlockBaseAddr
FlexRam block base address property.

enumerator kFLASH_PropertyFlexRamTotalSize
FlexRam total size property.

typedef enum *_flash_protection_state* flash_prot_state_t
Enumeration for the three possible flash protection levels.

typedef union *_pflash_protection_status* pflash_prot_status_t
PFlash protection status.

typedef enum *_flash_execute_only_access_state* flash_xacc_state_t
Enumeration for the three possible flash execute access levels.

typedef enum *_flash_property_tag* flash_property_tag_t
Enumeration for various flash properties.

```
typedef struct _flash_config flash_config_t
```

Flash driver state information.

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

```
kStatus_FLASH_Success
```

```
kFLASH_ApiEraseKey
```

```
union _pflash_protection_status
```

```
#include <fsl_ftfx_flash.h> PFlash protection status.
```

Public Members

```
uint32_t protl
```

```
PROT[31:0] .
```

```
uint32_t proth
```

```
PROT[63:32].
```

```
uint8_t protsl
```

```
PROTS[7:0] .
```

```
uint8_t protsh
```

```
PROTS[15:8] .
```

```
uint8_t reserved[2]
```

```
struct _flash_config
```

```
#include <fsl_ftfx_flash.h> Flash driver state information.
```

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

2.16 Ftfx FLEXNVM Driver

```
status_t FLEXNVM_Init(flexnvm_config_t *config)
```

Initializes the global flash properties structure members.

This function checks and initializes the Flash module for the other Flash APIs.

Parameters

- config – Pointer to the storage for the driver runtime state.

Return values

- kStatus_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.
- kStatus_FTFx_PartitionStatusUpdateFailure – Failed to update the partition status.

status_t FLEXNVM_DflashErase(*flexnvm_config_t* *config, uint32_t start, uint32_t lengthInBytes, uint32_t key)

Erases the Dflash sectors encompassed by parameters passed into function.

This function erases the appropriate number of flash sectors based on the desired start address and length.

Parameters

- config – The pointer to the storage for the driver runtime state.
- start – The start address of the desired flash memory to be erased. The start address does not need to be sector-aligned but must be word-aligned.
- lengthInBytes – The length, given in bytes (not words or long-words) to be erased. Must be word-aligned.
- key – The value used to validate all flash erase APIs.

Return values

- kStatus_FTFx_Success – API was executed successfully; the appropriate number of data flash sectors based on the desired start address and length were erased successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_AlignmentError – The parameter is not aligned with the specified baseline.
- kStatus_FTFx_AddressError – The address is out of range.
- kStatus_FTFx_EraseKeyError – The API erase key is invalid.
- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.
- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.
- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.
- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLEXNVM_EraseAll(*flexnvm_config_t* *config, uint32_t key)

Erases entire flexnvm.

Parameters

- config – Pointer to the storage for the driver runtime state.
- key – A value used to validate all flash erase APIs.

Return values

- kStatus_FTFx_Success – API was executed successfully; the entire flexnvm has been erased successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_EraseKeyError – API erase key is invalid.
- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.
- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.
- `kStatus_FTFx_PartitionStatusUpdateFailure` – Failed to update the partition status.

`status_t` FLEXNVM_EraseAllUnsecure(*flexnvm_config_t* *config, `uint32_t` key)

Erases the entire flexnvm, including protected sectors.

Parameters

- config – Pointer to the storage for the driver runtime state.
- key – A value used to validate all flash erase APIs.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the flexnvm is not in security state.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_EraseKeyError` – API erase key is invalid.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.
- `kStatus_FTFx_PartitionStatusUpdateFailure` – Failed to update the partition status.

`status_t` FLEXNVM_DflashProgram(*flexnvm_config_t* *config, `uint32_t` start, `uint8_t` *src, `uint32_t` lengthInBytes)

Programs flash with data at locations passed in through parameters.

This function programs the flash memory with the desired data for a given flash area as determined by the start address and the length.

Parameters

- config – A pointer to the storage for the driver runtime state.
- start – The start address of the desired flash memory to be programmed. Must be word-aligned.
- src – A pointer to the source buffer of data that is to be programmed into the flash.
- lengthInBytes – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the desired data have been successfully programed into specified date flash region.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with the specified baseline.

- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t` FLEXNVM_DflashProgramSection(*flexnvm_config_t* *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)

Programs flash with data at locations passed in through parameters via the Program Section command.

This function programs the flash memory with the desired data for a given flash area as determined by the start address and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be programmed. Must be word-aligned.
- `src` – A pointer to the source buffer of data that is to be programmed into the flash.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the desired data have been successfully programmed into specified data flash area.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FTFx_SetFlexramAsRamError` – Failed to set flexram as RAM.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.
- `kStatus_FTFx_RecoverFlexramAsEepromError` – Failed to recover FlexRAM as EEPROM.

`status_t` FLEXNVM_ProgramPartition(*flexnvm_config_t* *config, *ftfx_partition_flexram_load_opt_t* option, uint32_t eepromDataSizeCode, uint32_t flexnvmPartitionCode)

Prepares the FlexNVM block for use as data flash, EEPROM backup, or a combination of both and initializes the FlexRAM.

Parameters

- `config` – Pointer to storage for the driver runtime state.
- `option` – The option used to set FlexRAM load behavior during reset.
- `eeepromDataSizeCode` – Determines the amount of FlexRAM used in each of the available EEPROM subsystems.
- `flexnvmPartitionCode` – Specifies how to split the FlexNVM block between data flash memory and EEPROM backup memory supporting EEPROM functions.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the FlexNVM block for use as data flash, EEPROM backup, or a combination of both have been Prepared.
- `kStatus_FTFx_InvalidArgument` – Invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.

status_t FLEXNVM_ProgramPartition_CSE(*flexnvm_config_t* *config,
ftfx_partition_flexram_load_opt_t option, uint32_t
eeepromDataSizeCode, uint32_t
flexnvmPartitionCode, uint8_t CSEcKeySize, uint8_t
SFE)

Prepares the FlexNVM block for use as data flash, EEPROM backup, or a combination of both and initializes the FlexRAM. This is the CSE enabled version for IP's like FTFC.

Parameters

- `config` – Pointer to storage for the driver runtime state.
- `option` – The option used to set FlexRAM load behavior during reset.
- `eeepromDataSizeCode` – Determines the amount of FlexRAM used in each of the available EEPROM subsystems.
- `flexnvmPartitionCode` – Specifies how to split the FlexNVM block between data flash memory and EEPROM backup memory supporting EEPROM functions.
- `CSEcKeySize` – CSEc/SHE key size, see RM for details and possible values
- `SFE` – Security Flag Extension (SFE), see RM for details and possible values

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the FlexNVM block for use as data flash, EEPROM backup, or a combination of both have been Prepared.
- `kStatus_FTFx_InvalidArgument` – Invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.

- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.

`status_t FLEXNVM_ReadResource(flexnvm_config_t *config, uint32_t start, uint8_t *dst, uint32_t lengthInBytes, ftfx_read_resource_opt_t option)`

Reads the resource with data at locations passed in through parameters.

This function reads the flash memory with the desired location for a given flash area as determined by the start address and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be programmed. Must be word-aligned.
- `dst` – A pointer to the destination buffer of data that is used to store data to be read.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be read. Must be word-aligned.
- `option` – The resource option which indicates which area should be read back.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the data have been read successfully from program flash IFR, data flash IFR space, and the Version ID field
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with the specified baseline.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FLEXNVM_DflashVerifyErase(flexnvm_config_t *config, uint32_t start, uint32_t lengthInBytes, ftfx_margin_value_t margin)`

Verifies an erasure of the desired flash area at a specified margin level.

This function checks the appropriate number of flash sectors based on the desired start address and length to check whether the flash is erased to the specified read margin level.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be verified. The start address does not need to be sector-aligned but must be word-aligned.

- `lengthInBytes` – The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
- `margin` – Read margin choice.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the specified data flash region is in erased state.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FLEXNVM_VerifyEraseAll(flexnvm_config_t *config, ftfx_margin_value_t margin)`

Verifies erasure of the entire flash at a specified margin level.

This function checks whether the flash is erased to the specified read margin level.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `margin` – Read margin choice.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the entire flexnvm region is in erased state.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FLEXNVM_DflashVerifyProgram(flexnvm_config_t *config, uint32_t start, uint32_t lengthInBytes, const uint8_t *expectedData, ftfx_margin_value_t margin, uint32_t *failedAddress, uint32_t *failedData)`

Verifies programming of the desired flash area at a specified margin level.

This function verifies the data programmed in the flash memory using the Flash Program Check Command and compares it to the expected data for a given flash area as determined by the start address and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be verified. Must be word-aligned.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
- `expectedData` – A pointer to the expected data that is to be verified against.
- `margin` – Read margin choice.
- `failedAddress` – A pointer to the returned failing address.
- `failedData` – A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the desired data have been programmed successfully into specified data flash region.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t` FLEXNVM_GetSecurityState(*flexnvm_config_t* *config, *ftfx_security_state_t* *state)

Returns the security state via the pointer passed into the function.

This function retrieves the current flash security status, including the security enabling state and the backdoor key enabling state.

Parameters

- `config` – A pointer to storage for the driver runtime state.
- `state` – A pointer to the value returned for the current security status code:

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the security state of flexnvm was stored to state.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.

`status_t` FLEXNVM_SecurityBypass(*flexnvm_config_t* *config, const `uint8_t` *backdoorKey)

Allows users to bypass security with a backdoor key.

If the MCU is in secured state, this function unsecures the MCU by comparing the provided backdoor key with ones in the flash configuration field.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `backdoorKey` – A pointer to the user buffer containing the backdoor key.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FLEXNVM_SetFlexramFunction(flexnvm_config_t *config, ftfx_flexram_func_opt_t option)`

Sets the FlexRAM function command.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `option` – The option used to set the work mode of FlexRAM.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the FlexRAM has been successfully configured as RAM or EEPROM
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FTFx_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FTFx_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FTFx_CommandFailure` – Run-time error during the command execution.

`status_t FLEXNVM_DflashSetProtection(flexnvm_config_t *config, uint8_t protectStatus)`

Sets the DFlash protection to the intended protection status.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `protectStatus` – The expected protect status to set to the DFlash protection register. Each bit corresponds to the protection of the 1/8 of the total DFlash. The least significant bit corresponds to the lowest address area of the DFlash. The most significant bit corresponds to the highest address area of the DFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

- `kStatus_FTFx_Success` – API was executed successfully; the specified DFlash region is protected.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_CommandNotSupported` – Flash API is not supported.

- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.

`status_t` FLEXNVM_DflashGetProtection(*flexnvm_config_t* *config, `uint8_t` *protectStatus)

Gets the DFlash protection status.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `protectStatus` – DFlash Protect status returned by the PFlash IP. Each bit corresponds to the protection of the 1/8 of the total DFlash. The least significant bit corresponds to the lowest address area of the DFlash. The most significant bit corresponds to the highest address area of the DFlash, and so on. There are two possible cases as below: 0: this area is protected. 1: this area is unprotected.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_CommandNotSupported` – Flash API is not supported.

`status_t` FLEXNVM_EepromSetProtection(*flexnvm_config_t* *config, `uint8_t` protectStatus)

Sets the EEPROM protection to the intended protection status.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `protectStatus` – The expected protect status to set to the EEPROM protection register. Each bit corresponds to the protection of the 1/8 of the total EEPROM. The least significant bit corresponds to the lowest address area of the EEPROM. The most significant bit corresponds to the highest address area of EEPROM, and so on. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_CommandNotSupported` – Flash API is not supported.
- `kStatus_FTFx_CommandFailure` – Run-time error during command execution.

`status_t` FLEXNVM_EepromGetProtection(*flexnvm_config_t* *config, `uint8_t` *protectStatus)

Gets the EEPROM protection status.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `protectStatus` – DFlash Protect status returned by the PFlash IP. Each bit corresponds to the protection of the 1/8 of the total EEPROM. The least significant bit corresponds to the lowest address area of the EEPROM. The most significant bit corresponds to the highest address area of the EEPROM. There are two possible cases as below: 0: this area is protected. 1: this area is unprotected.

Return values

- `kStatus_FTFx_Success` – API was executed successfully.
- `kStatus_FTFx_InvalidArgument` – An invalid argument is provided.

- kStatus_FTFx_CommandNotSupported – Flash API is not supported.

status_t FLEXNVM_GetProperty(*flexnvm_config_t* *config, *flexnvm_property_tag_t* whichProperty, *uint32_t* *value)

Returns the desired flexnvm property.

Parameters

- config – A pointer to the storage for the driver runtime state.
- whichProperty – The desired property from the list of properties in enum *flexnvm_property_tag_t*
- value – A pointer to the value returned for the desired flexnvm property.

Return values

- kStatus_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_UnknownProperty – An unknown property tag.

enum *_flexnvm_property_tag*

Enumeration for various flexnvm properties.

Values:

enumerator kFLEXNVM_PropertyDflashSectorSize
Dflash sector size property.

enumerator kFLEXNVM_PropertyDflashTotalSize
Dflash total size property.

enumerator kFLEXNVM_PropertyDflashBlockSize
Dflash block size property.

enumerator kFLEXNVM_PropertyDflashBlockCount
Dflash block count property.

enumerator kFLEXNVM_PropertyDflashBlockBaseAddr
Dflash block base address property.

enumerator kFLEXNVM_PropertyAliasDflashBlockBaseAddr
Dflash block base address Alias property.

enumerator kFLEXNVM_PropertyFlexRamBlockBaseAddr
FlexRam block base address property.

enumerator kFLEXNVM_PropertyFlexRamTotalSize
FlexRam total size property.

enumerator kFLEXNVM_PropertyEepromTotalSize
EEPROM total size property.

typedef enum *_flexnvm_property_tag* *flexnvm_property_tag_t*

Enumeration for various flexnvm properties.

typedef struct *_flexnvm_config* *flexnvm_config_t*

Flexnvm driver state information.

An instance of this structure is allocated by the user of the Flexnvm driver and passed into each of the driver APIs.

status_t FLEXNVM_EepromWrite(*flexnvm_config_t* *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)

Programs the EEPROM with data at locations passed in through parameters.

This function programs the emulated EEPROM with the desired data for a given flash area as determined by the start address and length.

Parameters

- config – A pointer to the storage for the driver runtime state.
- start – The start address of the desired flash memory to be programmed. Must be word-aligned.
- src – A pointer to the source buffer of data that is to be programmed into the flash.
- lengthInBytes – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- kStatus_FTFx_Success – API was executed successfully; the desired data have been successfully programmed into specified eeprom region.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_AddressError – Address is out of range.
- kStatus_FTFx_SetFlexramAsEepromError – Failed to set flexram as eeprom.
- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.
- kStatus_FTFx_RecoverFlexramAsRamError – Failed to recover the FlexRAM as RAM.

struct _flexnvm_config

#include <fsl_ftfx_flexnvm.h> Flexnvm driver state information.

An instance of this structure is allocated by the user of the Flexnvm driver and passed into each of the driver APIs.

2.17 ftfx utilities

ALIGN_DOWN(x, a)

Alignment(down) utility.

ALIGN_UP(x, a)

Alignment(up) utility.

MAKE_VERSION(major, minor, bugfix)

Constructs the version number for drivers.

MAKE_STATUS(group, code)

Constructs a status code value from a group and a code number.

FOUR_CHAR_CODE(a, b, c, d)

Constructs the four character code for the Flash driver API key.

B1P4(b)

bytes2word utility.

B1P3(b)

B1P2(b)

B1P1(b)

B2P3(b)

B2P2(b)

B2P1(b)

B3P2(b)

B3P1(b)

BYTE2WORD_1_3(x, y)

BYTE2WORD_2_2(x, y)

BYTE2WORD_3_1(x, y)

BYTE2WORD_1_1_2(x, y, z)

BYTE2WORD_1_2_1(x, y, z)

BYTE2WORD_2_1_1(x, y, z)

BYTE2WORD_1_1_1_1(x, y, z, w)

2.18 GPIO: General-Purpose Input/Output Driver

FSL_GPIO_DRIVER_VERSION

GPIO driver version.

enum _gpio_pin_direction

GPIO direction definition.

Values:

enumerator kGPIO_DigitalInput

Set current pin as digital input

enumerator kGPIO_DigitalOutput

Set current pin as digital output

enum _gpio_checker_attribute

GPIO checker attribute.

Values:

enumerator kGPIO_UsernonsecureRWUsersecureRWPrivilegedsecureRW

User nonsecure:Read+Write; User Secure:Read+Write; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureRUsersecureRWPrivilegedsecureRW

User nonsecure:Read; User Secure:Read+Write; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureNUsersecureRWPrivilegedsecureRW

User nonsecure:None; User Secure:Read+Write; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureRUsersecureRPrivilegedsecureRW

User nonsecure:Read; User Secure:Read; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureNUsersecureRPrivilegedsecureRW
User nonsecure:None; User Secure:Read; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureNUsersecureNPrivilegedsecureRW
User nonsecure:None; User Secure:None; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureNUsersecureNPrivilegedsecureR
User nonsecure:None; User Secure:None; Privileged Secure:Read

enumerator kGPIO_UsernonsecureNUsersecureNPrivilegedsecureN
User nonsecure:None; User Secure:None; Privileged Secure:None

enumerator kGPIO_IgnoreAttributeCheck
Ignores the attribute check

typedef enum *_gpio_pin_direction* gpio_pin_direction_t
GPIO direction definition.

typedef enum *_gpio_checker_attribute* gpio_checker_attribute_t
GPIO checker attribute.

typedef struct *_gpio_pin_config* gpio_pin_config_t
The GPIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, leave the outputConfig unused. Note that in some use cases, the corresponding port property should be configured in advance with the PORT_SetPinConfig().

GPIO_FIT_REG(value)

struct *_gpio_pin_config*
#include <fsl_gpio.h> The GPIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, leave the outputConfig unused. Note that in some use cases, the corresponding port property should be configured in advance with the PORT_SetPinConfig().

Public Members

gpio_pin_direction_t pinDirection
GPIO direction, input or output

uint8_t outputLogic
Set a default output logic, which has no use in input

2.19 GPIO Driver

void GPIO_PortInit(GPIO_Type *base)
Initializes the GPIO peripheral.
This function ungates the GPIO clock.

Parameters

- base – GPIO peripheral base pointer.

void GPIO_PortDeinit(GPIO_Type *base)
Denitalizes the GPIO peripheral.

Parameters

- base – GPIO peripheral base pointer.

void GPIO_PinInit(GPIO_Type *base, uint32_t pin, const *gpio_pin_config_t* *config)

Initializes a GPIO pin used by the board.

To initialize the GPIO, define a pin configuration, as either input or output, in the user file. Then, call the GPIO_PinInit() function.

This is an example to define an input pin or an output pin configuration.

```
Define a digital input pin configuration,
gpio_pin_config_t config =
{
    kGPIO_DigitalInput,
    0,
}
Define a digital output pin configuration,
gpio_pin_config_t config =
{
    kGPIO_DigitalOutput,
    0,
}
```

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- pin – GPIO port pin number
- config – GPIO pin configuration pointer

static inline void GPIO_PinWrite(GPIO_Type *base, uint32_t pin, uint8_t output)

Sets the output level of the multiple GPIO pins to the logic 1 or 0.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- pin – GPIO pin number
- output – GPIO pin output logic level.
 - 0: corresponding pin output low-logic level.
 - 1: corresponding pin output high-logic level.

static inline void GPIO_PortSet(GPIO_Type *base, uint32_t mask)

Sets the output level of the multiple GPIO pins to the logic 1.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_PortClear(GPIO_Type *base, uint32_t mask)

Sets the output level of the multiple GPIO pins to the logic 0.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_PortToggle(GPIO_Type *base, uint32_t mask)

Reverses the current output logic of the multiple GPIO pins.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)

- mask – GPIO pin number macro

static inline uint32_t GPIO_PinRead(GPIO_Type *base, uint32_t pin)

Reads the current input value of the GPIO port.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- pin – GPIO pin number

Return values

GPIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

uint32_t GPIO_PortGetInterruptFlags(GPIO_Type *base)

Reads the GPIO port interrupt status flag.

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)

Return values

The – current GPIO port interrupt status flag, for example, 0x00010001 means the pin 0 and 17 have the interrupt.

void GPIO_PortClearInterruptFlags(GPIO_Type *base, uint32_t mask)

Clears multiple GPIO pin interrupt status flags.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

void GPIO_CheckAttributeBytes(GPIO_Type *base, *gpio_checker_attribute_t* attribute)

The GPIO module supports a device-specific number of data ports, organized as 32-bit words/8-bit Bytes. Each 32-bit/8-bit data port includes a GACR register, which defines the byte-level attributes required for a successful access to the GPIO programming model. If the GPIO module's GACR register is organized as 32-bit words, the attribute controls for the 4 data bytes in the GACR follow a standard little endian data convention.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- attribute – GPIO checker attribute

2.20 I2C: Inter-Integrated Circuit Driver

2.21 I2C DMA Driver

void I2C_MasterTransferCreateHandleDMA(I2C_Type *base, *i2c_master_dma_handle_t* *handle, *i2c_master_dma_transfer_callback_t* callback, void *userData, *dma_handle_t* *dmaHandle)

Initializes the I2C handle which is used in transactional functions.

Parameters

- base – I2C peripheral base address
- handle – Pointer to the `i2c_master_dma_handle_t` structure
- callback – Pointer to the user callback function
- userData – A user parameter passed to the callback function
- dmaHandle – DMA handle pointer

`status_t I2C_MasterTransferDMA(I2C_Type *base, i2c_master_dma_handle_t *handle, i2c_master_transfer_t *xfer)`

Performs a master DMA non-blocking transfer on the I2C bus.

Parameters

- base – I2C peripheral base address
- handle – A pointer to the `i2c_master_dma_handle_t` structure
- xfer – A pointer to the transfer structure of the `i2c_master_transfer_t`

Return values

- `kStatus_Success` – Successfully completes the data transmission.
- `kStatus_I2C_Busy` – A previous transmission is still not finished.
- `kStatus_I2C_Timeout` – A transfer error, waits for the signal timeout.
- `kStatus_I2C_ArbitrationLost` – A transfer error, arbitration lost.
- `kStatus_I2C_Nak` – A transfer error, receives NAK during transfer.

`status_t I2C_MasterTransferGetCountDMA(I2C_Type *base, i2c_master_dma_handle_t *handle, size_t *count)`

Gets a master transfer status during a DMA non-blocking transfer.

Parameters

- base – I2C peripheral base address
- handle – A pointer to the `i2c_master_dma_handle_t` structure
- count – A number of bytes transferred so far by the non-blocking transaction.

`void I2C_MasterTransferAbortDMA(I2C_Type *base, i2c_master_dma_handle_t *handle)`

Aborts a master DMA non-blocking transfer early.

Parameters

- base – I2C peripheral base address
- handle – A pointer to the `i2c_master_dma_handle_t` structure.

`FSL_I2C_DMA_DRIVER_VERSION`

I2C DMA driver version.

`typedef struct i2c_master_dma_handle i2c_master_dma_handle_t`

Retry times for waiting flag.

I2C master DMA handle typedef.

`typedef void (*i2c_master_dma_transfer_callback_t)(I2C_Type *base, i2c_master_dma_handle_t *handle, status_t status, void *userData)`

I2C master DMA transfer callback typedef.

`struct i2c_master_dma_handle`

`#include <fsl_i2c_dma.h>` I2C master DMA transfer structure.

Public Members

i2c_master_transfer_t transfer

I2C master transfer struct.

size_t transferSize

Total bytes to be transferred.

uint8_t state

I2C master transfer status.

dma_handle_t *dmaHandle

The DMA handler used.

i2c_master_dma_transfer_callback_t completionCallback

A callback function called after the DMA transfer finished.

void *userData

A callback parameter passed to the callback function.

2.22 I2C Driver

void I2C_MasterInit(I2C_Type *base, const *i2c_master_config_t* *masterConfig, uint32_t srcClock_Hz)

Initializes the I2C peripheral. Call this API to ungate the I2C clock and configure the I2C with master configuration.

Note: This API should be called at the beginning of the application. Otherwise, any operation to the I2C module can cause a hard fault because the clock is not enabled. The configuration structure can be custom filled or it can be set with default values by using the I2C_MasterGetDefaultConfig(). After calling this API, the master is ready to transfer. This is an example.

```
i2c_master_config_t config = {  
    .enableMaster = true,  
    .enableStopHold = false,  
    .highDrive = false,  
    .baudRate_Bps = 100000,  
    .glitchFilterWidth = 0  
};  
I2C_MasterInit(I2C0, &config, 12000000U);
```

Parameters

- base – I2C base pointer
- masterConfig – A pointer to the master configuration structure
- srcClock_Hz – I2C peripheral clock frequency in Hz

void I2C_SlaveInit(I2C_Type *base, const *i2c_slave_config_t* *slaveConfig, uint32_t srcClock_Hz)

Initializes the I2C peripheral. Call this API to ungate the I2C clock and initialize the I2C with the slave configuration.

Note: This API should be called at the beginning of the application. Otherwise, any operation to the I2C module can cause a hard fault because the clock is not enabled. The configuration structure can partly be set with default values by I2C_SlaveGetDefaultConfig() or it can be custom filled by the user. This is an example.

```
i2c_slave_config_t config = {
    .enableSlave = true,
    .enableGeneralCall = false,
    .addressingMode = kI2C_Address7bit,
    .slaveAddress = 0x1DU,
    .enableWakeUp = false,
    .enablehighDrive = false,
    .enableBaudRateCtl = false,
    .sclStopHoldTime_ns = 4000
};
I2C_SlaveInit(I2C0, &config, 12000000U);
```

Parameters

- base – I2C base pointer
- slaveConfig – A pointer to the slave configuration structure
- srcClock_Hz – I2C peripheral clock frequency in Hz

void I2C_MasterDeinit(I2C_Type *base)

De-initializes the I2C master peripheral. Call this API to gate the I2C clock. The I2C master module can't work unless the I2C_MasterInit is called.

Parameters

- base – I2C base pointer

void I2C_SlaveDeinit(I2C_Type *base)

De-initializes the I2C slave peripheral. Calling this API gates the I2C clock. The I2C slave module can't work unless the I2C_SlaveInit is called to enable the clock.

Parameters

- base – I2C base pointer

uint32_t I2C_GetInstance(I2C_Type *base)

Get instance number for I2C module.

Parameters

- base – I2C peripheral base address.

void I2C_MasterGetDefaultConfig(*i2c_master_config_t* *masterConfig)

Sets the I2C master configuration structure to default values.

The purpose of this API is to get the configuration structure initialized for use in the I2C_MasterConfigure(). Use the initialized structure unchanged in the I2C_MasterConfigure() or modify the structure before calling the I2C_MasterConfigure(). This is an example.

```
i2c_master_config_t config;
I2C_MasterGetDefaultConfig(&config);
```

Parameters

- masterConfig – A pointer to the master configuration structure.

void I2C_SlaveGetDefaultConfig(*i2c_slave_config_t* *slaveConfig)

Sets the I2C slave configuration structure to default values.

The purpose of this API is to get the configuration structure initialized for use in the I2C_SlaveConfigure(). Modify fields of the structure before calling the I2C_SlaveConfigure(). This is an example.

```
i2c_slave_config_t config;
I2C_SlaveGetDefaultConfig(&config);
```

Parameters

- slaveConfig – A pointer to the slave configuration structure.

```
static inline void I2C_Enable(I2C_Type *base, bool enable)
```

Enables or disables the I2C peripheral operation.

Parameters

- base – I2C base pointer
- enable – Pass true to enable and false to disable the module.

```
uint32_t I2C_MasterGetStatusFlags(I2C_Type *base)
```

Gets the I2C status flags.

Parameters

- base – I2C base pointer

Returns

status flag, use status flag to AND `_i2c_flags` to get the related status.

```
static inline uint32_t I2C_SlaveGetStatusFlags(I2C_Type *base)
```

Gets the I2C status flags.

Parameters

- base – I2C base pointer

Returns

status flag, use status flag to AND `_i2c_flags` to get the related status.

```
static inline void I2C_MasterClearStatusFlags(I2C_Type *base, uint32_t statusMask)
```

Clears the I2C status flag state.

The following status register flags can be cleared `kI2C_ArbitrationLostFlag` and `kI2C_IntPendingFlag`.

Parameters

- base – I2C base pointer
- statusMask – The status flag mask, defined in type `i2c_status_flag_t`. The parameter can be any combination of the following values:
 - `kI2C_StartDetectFlag` (if available)
 - `kI2C_StopDetectFlag` (if available)
 - `kI2C_ArbitrationLostFlag`
 - `kI2C_IntPendingFlag`

```
static inline void I2C_SlaveClearStatusFlags(I2C_Type *base, uint32_t statusMask)
```

Clears the I2C status flag state.

The following status register flags can be cleared `kI2C_ArbitrationLostFlag` and `kI2C_IntPendingFlag`

Parameters

- base – I2C base pointer
- statusMask – The status flag mask, defined in type `i2c_status_flag_t`. The parameter can be any combination of the following values:

- kI2C_StartDetectFlag (if available)
- kI2C_StopDetectFlag (if available)
- kI2C_ArbitrationLostFlag
- kI2C_IntPendingFlagFlag

void I2C__EnableInterrupts(I2C_Type *base, uint32_t mask)

Enables I2C interrupt requests.

Parameters

- base – I2C base pointer
- mask – interrupt source The parameter can be combination of the following source if defined:
 - kI2C_GlobalInterruptEnable
 - kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable
 - kI2C_SdaTimeoutInterruptEnable

void I2C__DisableInterrupts(I2C_Type *base, uint32_t mask)

Disables I2C interrupt requests.

Parameters

- base – I2C base pointer
- mask – interrupt source The parameter can be combination of the following source if defined:
 - kI2C_GlobalInterruptEnable
 - kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable
 - kI2C_SdaTimeoutInterruptEnable

static inline void I2C__EnableDMA(I2C_Type *base, bool enable)

Enables/disables the I2C DMA interrupt.

Parameters

- base – I2C base pointer
- enable – true to enable, false to disable

static inline uint32_t I2C__GetDataRegAddr(I2C_Type *base)

Gets the I2C tx/rx data register address. This API is used to provide a transfer address for I2C DMA transfer configuration.

Parameters

- base – I2C base pointer

Returns

data register address

void I2C__MasterSetBaudRate(I2C_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)

Sets the I2C master transfer baud rate.

Parameters

- base – I2C base pointer
- baudRate_Bps – the baud rate value in bps
- srcClock_Hz – Source clock

status_t I2C_MasterStart(I2C_Type *base, uint8_t address, i2c_direction_t direction)

Sends a START on the I2C bus.

This function is used to initiate a new master mode transfer by sending the START signal. The slave address is sent following the I2C START signal.

Parameters

- base – I2C peripheral base pointer
- address – 7-bit slave device address.
- direction – Master transfer directions(transmit/receive).

Return values

- kStatus_Success – Successfully send the start signal.
- kStatus_I2C_Busy – Current bus is busy.

status_t I2C_MasterStop(I2C_Type *base)

Sends a STOP signal on the I2C bus.

Return values

- kStatus_Success – Successfully send the stop signal.
- kStatus_I2C_Timeout – Send stop signal failed, timeout.

status_t I2C_MasterRepeatedStart(I2C_Type *base, uint8_t address, i2c_direction_t direction)

Sends a REPEATED START on the I2C bus.

Parameters

- base – I2C peripheral base pointer
- address – 7-bit slave device address.
- direction – Master transfer directions(transmit/receive).

Return values

- kStatus_Success – Successfully send the start signal.
- kStatus_I2C_Busy – Current bus is busy but not occupied by current I2C master.

status_t I2C_MasterWriteBlocking(I2C_Type *base, const uint8_t *txBuff, size_t txSize, uint32_t flags)

Performs a polling send transaction on the I2C bus.

Parameters

- base – The I2C peripheral base pointer.
- txBuff – The pointer to the data to be transferred.
- txSize – The length in bytes of the data to be transferred.
- flags – Transfer control flag to decide whether need to send a stop, use kI2C_TransferDefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop.

Return values

- kStatus_Success – Successfully complete the data transmission.
- kStatus_I2C_ArbitrationLost – Transfer error, arbitration lost.
- kStatus_I2C_Nak – Transfer error, receive NAK during transfer.

status_t I2C_MasterReadBlocking(I2C_Type *base, uint8_t *rxBuff, size_t rxSize, uint32_t flags)

Performs a polling receive transaction on the I2C bus.

Note: The I2C_MasterReadBlocking function stops the bus before reading the final byte. Without stopping the bus prior for the final read, the bus issues another read, resulting in garbage data being read into the data register.

Parameters

- base – I2C peripheral base pointer.
- rxBuff – The pointer to the data to store the received data.
- rxSize – The length in bytes of the data to be received.
- flags – Transfer control flag to decide whether need to send a stop, use kI2C_TransferDefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop.

Return values

- kStatus_Success – Successfully complete the data transmission.
- kStatus_I2C_Timeout – Send stop signal failed, timeout.

status_t I2C_SlaveWriteBlocking(I2C_Type *base, const uint8_t *txBuff, size_t txSize)

Performs a polling send transaction on the I2C bus.

Parameters

- base – The I2C peripheral base pointer.
- txBuff – The pointer to the data to be transferred.
- txSize – The length in bytes of the data to be transferred.

Return values

- kStatus_Success – Successfully complete the data transmission.
- kStatus_I2C_ArbitrationLost – Transfer error, arbitration lost.
- kStatus_I2C_Nak – Transfer error, receive NAK during transfer.

status_t I2C_SlaveReadBlocking(I2C_Type *base, uint8_t *rxBuff, size_t rxSize)

Performs a polling receive transaction on the I2C bus.

Parameters

- base – I2C peripheral base pointer.
- rxBuff – The pointer to the data to store the received data.
- rxSize – The length in bytes of the data to be received.

Return values

- kStatus_Success – Successfully complete data receive.
- kStatus_I2C_Timeout – Wait status flag timeout.

status_t I2C_MasterTransferBlocking(I2C_Type *base, i2c_master_transfer_t *xfer)

Performs a master polling transfer on the I2C bus.

Note: The API does not return until the transfer succeeds or fails due to arbitration lost or receiving a NAK.

Parameters

- base – I2C peripheral base address.
- xfer – Pointer to the transfer structure.

Return values

- kStatus_Success – Successfully complete the data transmission.
- kStatus_I2C_Busy – Previous transmission still not finished.
- kStatus_I2C_Timeout – Transfer error, wait signal timeout.
- kStatus_I2C_ArbitrationLost – Transfer error, arbitration lost.
- kStatus_I2C_Nak – Transfer error, receive NAK during transfer.

```
void I2C_MasterTransferCreateHandle(I2C_Type *base, i2c_master_handle_t *handle,  
                                   i2c_master_transfer_callback_t callback, void *userData)
```

Initializes the I2C handle which is used in transactional functions.

Parameters

- base – I2C base pointer.
- handle – pointer to i2c_master_handle_t structure to store the transfer state.
- callback – pointer to user callback function.
- userData – user parameter passed to the callback function.

```
status_t I2C_MasterTransferNonBlocking(I2C_Type *base, i2c_master_handle_t *handle,  
                                       i2c_master_transfer_t *xfer)
```

Performs a master interrupt non-blocking transfer on the I2C bus.

Note: Calling the API returns immediately after transfer initiates. The user needs to call I2C_MasterGetTransferCount to poll the transfer status to check whether the transfer is finished. If the return status is not kStatus_I2C_Busy, the transfer is finished.

Parameters

- base – I2C base pointer.
- handle – pointer to i2c_master_handle_t structure which stores the transfer state.
- xfer – pointer to i2c_master_transfer_t structure.

Return values

- kStatus_Success – Successfully start the data transmission.
- kStatus_I2C_Busy – Previous transmission still not finished.
- kStatus_I2C_Timeout – Transfer error, wait signal timeout.

```
status_t I2C_MasterTransferGetCount(I2C_Type *base, i2c_master_handle_t *handle, size_t  
                                   *count)
```

Gets the master transfer status during a interrupt non-blocking transfer.

Parameters

- base – I2C base pointer.
- handle – pointer to i2c_master_handle_t structure which stores the transfer state.

- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_Success` – Successfully return the count.

`status_t I2C_MasterTransferAbort(I2C_Type *base, i2c_master_handle_t *handle)`

Aborts an interrupt non-blocking transfer early.

Note: This API can be called at any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- `base` – I2C base pointer.
- `handle` – pointer to `i2c_master_handle_t` structure which stores the transfer state

Return values

- `kStatus_I2C_Timeout` – Timeout during polling flag.
- `kStatus_Success` – Successfully abort the transfer.

`void I2C_MasterTransferHandleIRQ(I2C_Type *base, void *i2cHandle)`

Master interrupt handler.

Parameters

- `base` – I2C base pointer.
- `i2cHandle` – pointer to `i2c_master_handle_t` structure.

`void I2C_SlaveTransferCreateHandle(I2C_Type *base, i2c_slave_handle_t *handle, i2c_slave_transfer_callback_t callback, void *userData)`

Initializes the I2C handle which is used in transactional functions.

Parameters

- `base` – I2C base pointer.
- `handle` – pointer to `i2c_slave_handle_t` structure to store the transfer state.
- `callback` – pointer to user callback function.
- `userData` – user parameter passed to the callback function.

`status_t I2C_SlaveTransferNonBlocking(I2C_Type *base, i2c_slave_handle_t *handle, uint32_t eventMask)`

Starts accepting slave transfers.

Call this API after calling the `I2C_SlaveInit()` and `I2C_SlaveTransferCreateHandle()` to start processing transactions driven by an I2C master. The slave monitors the I2C bus and passes events to the callback that was passed into the call to `I2C_SlaveTransferCreateHandle()`. The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the `eventMask` parameter to the OR'd combination of `i2c_slave_transfer_event_t` enumerators for the events you wish to receive. The `kI2C_SlaveTransmitEvent` and `kLPI2C_SlaveReceiveEvent` events are always enabled and do not need to be included in the mask. Alternatively, pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the `kI2C_SlaveAllEvents` constant is provided as a convenient way to enable all events.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to `i2c_slave_handle_t` structure which stores the transfer state.
- `eventMask` – Bit mask formed by OR'ing together `i2c_slave_transfer_event_t` enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and `kI2C_SlaveAllEvents` to enable all events.

Return values

- `kStatus_Success` – Slave transfers were successfully started.
- `kStatus_I2C_Busy` – Slave transfers have already been started on this handle.

`void I2C_SlaveTransferAbort(I2C_Type *base, i2c_slave_handle_t *handle)`

Aborts the slave transfer.

Note: This API can be called at any time to stop slave for handling the bus events.

Parameters

- `base` – I2C base pointer.
- `handle` – pointer to `i2c_slave_handle_t` structure which stores the transfer state.

`status_t I2C_SlaveTransferGetCount(I2C_Type *base, i2c_slave_handle_t *handle, size_t *count)`

Gets the slave transfer remaining bytes during a interrupt non-blocking transfer.

Parameters

- `base` – I2C base pointer.
- `handle` – pointer to `i2c_slave_handle_t` structure.
- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_Success` – Successfully return the count.

`void I2C_SlaveTransferHandleIRQ(I2C_Type *base, void *i2cHandle)`

Slave interrupt handler.

Parameters

- `base` – I2C base pointer.
- `i2cHandle` – pointer to `i2c_slave_handle_t` structure which stores the transfer state

`FSL_I2C_DRIVER_VERSION`

I2C driver version.

I2C status return codes.

Values:

enumerator `kStatus_I2C_Busy`

I2C is busy with current transfer.

enumerator kStatus_I2C_Idle

Bus is Idle.

enumerator kStatus_I2C_Nak

NAK received during transfer.

enumerator kStatus_I2C_ArbitrationLost

Arbitration lost during transfer.

enumerator kStatus_I2C_Timeout

Timeout polling status flags.

enumerator kStatus_I2C_Addr_Nak

NAK received during the address probe.

enum _i2c_flags

I2C peripheral flags.

Note: These enumerations are meant to be OR'd together to form a bit mask.

Values:

enumerator kI2C_ReceiveNakFlag

I2C receive NAK flag.

enumerator kI2C_IntPendingFlag

I2C interrupt pending flag. This flag can be cleared.

enumerator kI2C_TransferDirectionFlag

I2C transfer direction flag.

enumerator kI2C_RangeAddressMatchFlag

I2C range address match flag.

enumerator kI2C_ArbitrationLostFlag

I2C arbitration lost flag. This flag can be cleared.

enumerator kI2C_BusBusyFlag

I2C bus busy flag.

enumerator kI2C_AddressMatchFlag

I2C address match flag.

enumerator kI2C_TransferCompleteFlag

I2C transfer complete flag.

enumerator kI2C_StopDetectFlag

I2C stop detect flag. This flag can be cleared.

enumerator kI2C_StartDetectFlag

I2C start detect flag. This flag can be cleared.

enum _i2c_interrupt_enable

I2C feature interrupt source.

Values:

enumerator kI2C_GlobalInterruptEnable

I2C global interrupt.

enumerator kI2C_StopDetectInterruptEnable

I2C stop detect interrupt.

enumerator kI2C_StartStopDetectInterruptEnable
I2C start&stop detect interrupt.

enum _i2c_direction
The direction of master and slave transfers.

Values:

enumerator kI2C_Write
Master transmits to the slave.

enumerator kI2C_Read
Master receives from the slave.

enum _i2c_slave_address_mode
Addressing mode.

Values:

enumerator kI2C_Address7bit
7-bit addressing mode.

enumerator kI2C_RangeMatch
Range address match addressing mode.

enum _i2c_master_transfer_flags
I2C transfer control flag.

Values:

enumerator kI2C_TransferDefaultFlag
A transfer starts with a start signal, stops with a stop signal.

enumerator kI2C_TransferNoStartFlag
A transfer starts without a start signal, only support write only or write+read with no start flag, do not support read only with no start flag.

enumerator kI2C_TransferRepeatedStartFlag
A transfer starts with a repeated start signal.

enumerator kI2C_TransferNoStopFlag
A transfer ends without a stop signal.

enum _i2c_slave_transfer_event
Set of events sent to the callback for nonblocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to I2C_SlaveTransferNonBlocking() to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

Values:

enumerator kI2C_SlaveAddressMatchEvent
Received the slave address after a start or repeated start.

enumerator kI2C_SlaveTransmitEvent
A callback is requested to provide data to transmit (slave-transmitter role).

enumerator `kI2C_SlaveReceiveEvent`

A callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator `kI2C_SlaveTransmitAckEvent`

A callback needs to either transmit an ACK or NACK.

enumerator `kI2C_SlaveStartEvent`

A start/repeated start was detected.

enumerator `kI2C_SlaveCompletionEvent`

A stop was detected or finished transfer, completing the transfer.

enumerator `kI2C_SlaveGeneralCallEvent`

Received the general call address after a start or repeated start.

enumerator `kI2C_SlaveAllEvents`

A bit mask of all available events.

Common sets of flags used by the driver.

Values:

enumerator `kClearFlags`

All flags which are cleared by the driver upon starting a transfer.

enumerator `kIrqFlags`

typedef enum `_i2c_direction` `i2c_direction_t`

The direction of master and slave transfers.

typedef enum `_i2c_slave_address_mode` `i2c_slave_address_mode_t`

Addressing mode.

typedef enum `_i2c_slave_transfer_event` `i2c_slave_transfer_event_t`

Set of events sent to the callback for nonblocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to `I2C_SlaveTransferNonBlocking()` to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

typedef struct `_i2c_master_config` `i2c_master_config_t`

I2C master user configuration.

typedef struct `_i2c_slave_config` `i2c_slave_config_t`

I2C slave user configuration.

typedef struct `_i2c_master_handle` `i2c_master_handle_t`

I2C master handle typedef.

typedef void (*`i2c_master_transfer_callback_t`)(`I2C_Type` *base, `i2c_master_handle_t` *handle, `status_t` status, void *userData)

I2C master transfer callback typedef.

typedef struct `_i2c_slave_handle` `i2c_slave_handle_t`

I2C slave handle typedef.

```
typedef struct _i2c_master_transfer i2c_master_transfer_t  
    I2C master transfer structure.
```

```
typedef struct _i2c_slave_transfer i2c_slave_transfer_t  
    I2C slave transfer structure.
```

```
typedef void (*i2c_slave_transfer_callback_t)(I2C_Type *base, i2c_slave_transfer_t *xfer, void  
*userData)  
    I2C slave transfer callback typedef.
```

```
I2C_RETRY_TIMES  
    Retry times for waiting flag.
```

```
I2C_MASTER_FACK_CONTROL  
    Mater Fast ack control, control if master needs to manually write ack, this is used to low  
the speed of transfer for SoCs with feature FSL_FEATURE_I2C_HAS_DOUBLE_BUFFERING.
```

```
I2C_HAS_STOP_DETECT
```

```
struct _i2c_master_config  
    #include <fsl_i2c.h> I2C master user configuration.
```

Public Members

```
bool enableMaster  
    Enables the I2C peripheral at initialization time.
```

```
bool enableStopHold  
    Controls the stop hold enable.
```

```
bool enableDoubleBuffering  
    Controls double buffer enable; notice that enabling the double buffer disables the clock  
stretch.
```

```
uint32_t baudRate_Bps  
    Baud rate configuration of I2C peripheral.
```

```
uint8_t glitchFilterWidth  
    Controls the width of the glitch.
```

```
struct _i2c_slave_config  
    #include <fsl_i2c.h> I2C slave user configuration.
```

Public Members

```
bool enableSlave  
    Enables the I2C peripheral at initialization time.
```

```
bool enableGeneralCall  
    Enables the general call addressing mode.
```

```
bool enableWakeUp  
    Enables/disables waking up MCU from low-power mode.
```

```
bool enableDoubleBuffering  
    Controls a double buffer enable; notice that enabling the double buffer disables the  
clock stretch.
```

```
bool enableBaudRateCtl  
    Enables/disables independent slave baud rate on SCL in very fast I2C modes.
```

uint16_t slaveAddress

A slave address configuration.

uint16_t upperAddress

A maximum boundary slave address used in a range matching mode.

i2c_slave_address_mode_t addressingMode

An addressing mode configuration of i2c_slave_address_mode_config_t.

uint32_t sclStopHoldTime_ns

the delay from the rising edge of SCL (I2C clock) to the rising edge of SDA (I2C data) while SCL is high (stop condition), SDA hold time and SCL start hold time are also configured according to the SCL stop hold time.

struct __i2c_master_transfer

#include <fsl_i2c.h> I2C master transfer structure.

Public Members

uint32_t flags

A transfer flag which controls the transfer.

uint8_t slaveAddress

7-bit slave address.

i2c_direction_t direction

A transfer direction, read or write.

uint32_t subaddress

A sub address. Transferred MSB first.

uint8_t subaddressSize

A size of the command buffer.

uint8_t *volatile data

A transfer buffer.

volatile size_t dataSize

A transfer size.

struct __i2c_master_handle

#include <fsl_i2c.h> I2C master handle structure.

Public Members

i2c_master_transfer_t transfer

I2C master transfer copy.

size_t transferSize

Total bytes to be transferred.

uint8_t state

A transfer state maintained during transfer.

i2c_master_transfer_callback_t completionCallback

A callback function called when the transfer is finished.

void *userData

A callback parameter passed to the callback function.

struct __i2c_slave_transfer

#include <fsl_i2c.h> I2C slave transfer structure.

Public Members

i2c_slave_transfer_event_t event

A reason that the callback is invoked.

uint8_t *volatile data

A transfer buffer.

volatile size_t dataSize

A transfer size.

status_t completionStatus

Success or error code describing how the transfer completed. Only applies for kI2C_SlaveCompletionEvent.

size_t transferredCount

A number of bytes actually transferred since the start or since the last repeated start.

struct _i2c_slave_handle

#include <fsl_i2c.h> I2C slave handle structure.

Public Members

volatile bool isBusy

Indicates whether a transfer is busy.

i2c_slave_transfer_t transfer

I2C slave transfer copy.

uint32_t eventMask

A mask of enabled events.

i2c_slave_transfer_callback_t callback

A callback function called at the transfer event.

void *userData

A callback parameter passed to the callback.

2.23 IRTC: IRTC Driver

status_t IRTC_Init(RTC_Type *base, const *irtc_config_t* *config)

Ungates the IRTC clock and configures the peripheral for basic operation.

This function initiates a soft-reset of the IRTC module, this has not effect on DST, calendar-ing, standby time and tamper detect registers.

Note: This API should be called at the beginning of the application using the IRTC driver.

Parameters

- base – IRTC peripheral base address
- config – Pointer to user's IRTC config structure.

Returns

kStatus_Success If the driver is initialized successfully.

Returns

kStatus_Fail if we cannot disable register write protection

Returns

kStatus_InvalidArgument If the input parameters are wrong.

status_t IRTC_Deinit(RTC_Type *base)

Gate the IRTC clock.

Parameters

- base – IRTC peripheral base address

Returns

kStatus_Success If the driver is initialized successfully.

Returns

kStatus_InvalidArgument If the input parameters are wrong.

void IRTC_GetDefaultConfig(*irtc_config_t* *config)

Fill in the IRTC config struct with the default settings.

The default values are:

```
config->wakeupSelect = true;
config->timerStdMask = false;
config->alarmMatch = kRTC_MatchSecMinHr;
```

Parameters

- config – Pointer to user's IRTC config structure.

status_t IRTC_SetDatetime(RTC_Type *base, const *irtc_datetime_t* *datetime)

Sets the IRTC date and time according to the given time structure.

The IRTC counter is started after the time is set.

Parameters

- base – IRTC peripheral base address
- datetime – Pointer to structure where the date and time details to set are stored

Returns

kStatus_Success: success in setting the time and starting the IRTC
kStatus_InvalidArgument: failure. An error occurs because the datetime format is incorrect.

void IRTC_GetDatetime(RTC_Type *base, *irtc_datetime_t* *datetime)

Gets the IRTC time and stores it in the given time structure.

Parameters

- base – IRTC peripheral base address
- datetime – Pointer to structure where the date and time details are stored.

status_t IRTC_SetAlarm(RTC_Type *base, const *irtc_datetime_t* *alarmTime)

Sets the IRTC alarm time.

Note: weekDay field of alarmTime is not used during alarm match and should be set to 0

Parameters

- base – RTC peripheral base address
- alarmTime – Pointer to structure where the alarm time is stored.

Returns

kStatus_Success: success in setting the alarm
kStatus_InvalidArgument: error in setting the alarm. Error occurs because the alarm datetime format is incorrect.

void IRTC_GetAlarm(RTC_Type *base, *irtc_datetime_t* *datetime)

Returns the IRTC alarm time.

Parameters

- base – RTC peripheral base address
- datetime – Pointer to structure where the alarm date and time details are stored.

static inline void IRTC_EnableInterrupts(RTC_Type *base, uint32_t mask)

Enables the selected IRTC interrupts.

Parameters

- base – IRTC peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration *irtc_interrupt_enable_t*

static inline void IRTC_DisableInterrupts(RTC_Type *base, uint32_t mask)

Disables the selected IRTC interrupts.

Parameters

- base – IRTC peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration *irtc_interrupt_enable_t*

static inline uint32_t IRTC_GetEnabledInterrupts(RTC_Type *base)

Gets the enabled IRTC interrupts.

Parameters

- base – IRTC peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration *irtc_interrupt_enable_t*

static inline uint32_t IRTC_GetStatusFlags(RTC_Type *base)

Gets the IRTC status flags.

Parameters

- base – IRTC peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration *irtc_status_flags_t*

static inline void IRTC_ClearStatusFlags(RTC_Type *base, uint32_t mask)

Clears the IRTC status flags.

Parameters

- base – IRTC peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration *irtc_status_flags_t*

```
void IRTC_SetDaylightTime(RTC_Type *base, const irtc_daylight_time_t *datetime)
```

Sets the IRTC daylight savings start and stop date and time.

It also enables the daylight saving bit in the IRTC control register

Parameters

- *base* – IRTC peripheral base address
- *datetime* – Pointer to a structure where the date and time details are stored.

```
void IRTC_GetDaylightTime(RTC_Type *base, irtc_daylight_time_t *datetime)
```

Gets the IRTC daylight savings time and stores it in the given time structure.

Parameters

- *base* – IRTC peripheral base address
- *datetime* – Pointer to a structure where the date and time details are stored.

```
void IRTC_SetCoarseCompensation(RTC_Type *base, uint8_t compensationValue, uint8_t  
                                compensationInterval)
```

Enables the coarse compensation and sets the value in the IRTC compensation register.

Parameters

- *base* – IRTC peripheral base address
- *compensationValue* – Compensation value is a 2's complement value.
- *compensationInterval* – Compensation interval.

```
void IRTC_SetFineCompensation(RTC_Type *base, uint8_t integralValue, uint8_t fractionValue,  
                              bool accumulateFractional)
```

Enables the fine compensation and sets the value in the IRTC compensation register.

Parameters

- *base* – The IRTC peripheral base address
- *integralValue* – Compensation integral value; twos complement value of the integer part
- *fractionValue* – Compensation fraction value expressed as number of clock cycles of a fixed 4.194304Mhz clock that have to be added.
- *accumulateFractional* – Flag indicating if we want to add to previous fractional part; true: Add to previously accumulated fractional part, false: Start afresh and overwrite current value

```
void IRTC_SetTamperParams(RTC_Type *base, irtc_tamper_pins_t tamperNumber, const  
                          irtc_tamper_config_t *tamperConfig)
```

This function allows configuring the four tamper inputs.

The function configures the filter properties for the three external tampers. It also sets up active/passive and direction of the tamper bits, which are not available on all platforms.

Note: This function programs the tamper filter parameters. The user must gate the 32K clock to the RTC before calling this function. It is assumed that the time and date are set after this and the tamper parameters do not require to be changed again later.

Parameters

- *base* – The IRTC peripheral base address
- *tamperNumber* – The IRTC tamper input to configure

- tamperConfig – The IRTC tamper properties

uint8_t IRTC_ReadTamperQueue(RTC_Type *base, irtc_datetime_t *tamperTimestamp)

This function reads the tamper timestamp and returns the associated tamper pin.

The tamper timestamp has month, day, hour, minutes, and seconds. Ignore the year field as this information is not available in the tamper queue. The user should look at the RTC_YEARMON register for this because the expectation is that the queue is read at least once a year. Return the tamper pin number associated with the timestamp.

Parameters

- base – The IRTC peripheral base address
- tamperTimestamp – The tamper timestamp

Returns

The tamper pin number

static inline bool IRTC_GetTamperQueueFullStatus(RTC_Type *base)

Gets the IRTC Tamper queue full status.

Parameters

- base – IRTC peripheral base address

Return values

- true – Tamper queue is full.
- false – Tamper queue is not full.

static inline void IRTC_ClearTamperQueueFullStatus(RTC_Type *base)

Clear the IRTC Tamper queue full status.

Parameters

- base – IRTC peripheral base address

FSL_IRTC_DRIVER_VERSION

enum _irtc_filter_clock_source

IRTC filter clock source options.

Values:

enumerator kIRTC_32K

Use 32 kHz clock source for the tamper filter.

enumerator kIRTC_512

Use 512 Hz clock source for the tamper filter.

enumerator kIRTC_128

Use 128 Hz clock source for the tamper filter.

enumerator kIRTC_64

Use 64 Hz clock source for the tamper filter.

enumerator kIRTC_16

Use 16 Hz clock source for the tamper filter.

enumerator kIRTC_8

Use 8 Hz clock source for the tamper filter.

enumerator kIRTC_4

Use 4 Hz clock source for the tamper filter.

enumerator kIRTC_2

Use 2 Hz clock source for the tamper filter.

enum __irtc_tamper_pins

IRTC Tamper pins.

Values:

enumerator kIRTC_Tamper_0

External Tamper 0

enumerator kIRTC_Tamper_1

External Tamper 1

enumerator kIRTC_Tamper_2

External Tamper 2

enumerator kIRTC_Tamper_3

Internal tamper, does not have filter configuration

enum __irtc_interrupt_enable

List of IRTC interrupts.

Values:

enumerator kIRTC_TamperInterruptEnable

Tamper Interrupt Enable

enumerator kIRTC_AlarmInterruptEnable

Alarm Interrupt Enable

enumerator kIRTC_DayInterruptEnable

Days Interrupt Enable

enumerator kIRTC_HourInterruptEnable

Hours Interrupt Enable

enumerator kIRTC_MinInterruptEnable

Minutes Interrupt Enable

enumerator kIRTC_1hzInterruptEnable

1 Hz interval Interrupt Enable

enumerator kIRTC_2hzInterruptEnable

2 Hz interval Interrupt Enable

enumerator kIRTC_4hzInterruptEnable

4 Hz interval Interrupt Enable

enumerator kIRTC_8hzInterruptEnable

8 Hz interval Interrupt Enable

enumerator kIRTC_16hzInterruptEnable

16 Hz interval Interrupt Enable

enumerator kIRTC_32hzInterruptEnable

32 Hz interval Interrupt Enable

enumerator kIRTC_64hzInterruptEnable

64 Hz interval Interrupt Enable

enumerator kIRTC_128hzInterruptEnable

128 Hz interval Interrupt Enable

enumerator kIRTC_256hzInterruptEnable
256 Hz interval Interrupt Enable

enumerator kIRTC_512hzInterruptEnable
512 Hz interval Interrupt Enable

enumerator kIRTC_TamperQueueFullInterruptEnable
Tamper queue full Interrupt Enable

enum _irtc_status_flags

List of IRTC flags.

Values:

enumerator kIRTC_TamperFlag
Tamper Status flag

enumerator kIRTC_AlarmFlag
Alarm Status flag

enumerator kIRTC_DayFlag
Days Status flag

enumerator kIRTC_HourFlag
Hour Status flag

enumerator kIRTC_MinFlag
Minutes Status flag

enumerator kIRTC_1hzFlag
1 Hz interval status flag

enumerator kIRTC_2hzFlag
2 Hz interval status flag

enumerator kIRTC_4hzFlag
4 Hz interval status flag

enumerator kIRTC_8hzFlag
8 Hz interval status flag

enumerator kIRTC_16hzFlag
16 Hz interval status flag

enumerator kIRTC_32hzFlag
32 Hz interval status flag

enumerator kIRTC_64hzFlag
64 Hz interval status flag

enumerator kIRTC_128hzFlag
128 Hz interval status flag

enumerator kIRTC_256hzFlag
256 Hz interval status flag

enumerator kIRTC_512hzFlag
512 Hz interval status flag

enumerator kIRTC_InvalidFlag
Indicates if time/date counters are invalid

enumerator kIRTC_WriteProtFlag
Write protect enable status flag

enumerator kIRTC_CpuLowVoltFlag
CPU low voltage warning flag

enumerator kIRTC_ResetSrcFlag
Reset source flag

enumerator kIRTC_CmpIntFlag
Compensation interval status flag

enumerator kIRTC_CmpDoneFlag
Compensation done flag

enumerator kIRTC_BusErrFlag
Bus error flag

enum _irtc_alarm_match
IRTC alarm match options.

Values:

enumerator kRTC_MatchSecMinHr
Only match second, minute and hour

enumerator kRTC_MatchSecMinHrDay
Only match second, minute, hour and day

enumerator kRTC_MatchSecMinHrDayMnth
Only match second, minute, hour, day and month

enumerator kRTC_MatchSecMinHrDayMnthYr
Only match second, minute, hour, day, month and year

enum _irtc_osc_cap_load
List of RTC Oscillator capacitor load settings.

Values:

enumerator kIRTC_Capacitor2p
2pF capacitor load

enumerator kIRTC_Capacitor4p
4pF capacitor load

enumerator kIRTC_Capacitor8p
8pF capacitor load

enumerator kIRTC_Capacitor16p
16pF capacitor load

enum _irtc_clockout_sel
IRTC clockout select.

Values:

enumerator kIRTC_ClkoutNo
No clock out

enumerator kIRTC_ClkoutFine1Hz
clock out fine 1Hz

enumerator kIRTC_Clkout32kHz
clock out 32.768kHz

enumerator kIRTC_ClkoutCoarse1Hz
clock out coarse 1Hz

typedef enum *_irtc_filter_clock_source* irtc_filter_clock_source_t
IRTC filter clock source options.

typedef enum *_irtc_tamper_pins* irtc_tamper_pins_t
IRTC Tamper pins.

typedef enum *_irtc_interrupt_enable* irtc_interrupt_enable_t
List of IRTC interrupts.

typedef enum *_irtc_status_flags* irtc_status_flags_t
List of IRTC flags.

typedef enum *_irtc_alarm_match* irtc_alarm_match_t
IRTC alarm match options.

typedef enum *_irtc_osc_cap_load* irtc_osc_cap_load_t
List of RTC Oscillator capacitor load settings.

typedef enum *_irtc_clockout_sel* irtc_clockout_sel_t
IRTC clockout select.

typedef struct *_irtc_datetime* irtc_datetime_t
Structure is used to hold the date and time.

typedef struct *_irtc_daylight_time* irtc_daylight_time_t
Structure is used to hold the daylight saving time.

typedef struct *_irtc_tamper_config* irtc_tamper_config_t
Structure is used to define the parameters to configure a RTC tamper event.

typedef struct *_irtc_config* irtc_config_t
RTC config structure.

This structure holds the configuration settings for the RTC peripheral. To initialize this structure to reasonable defaults, call the `IRTC_GetDefaultConfig()` function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

static inline void `IRTC_SetOscCapLoad(RTC_Type *base, uint16_t capLoad)`

This function sets the specified capacitor configuration for the RTC oscillator.

Parameters

- `base` – IRTC peripheral base address
- `capLoad` – Oscillator loads to enable. This is a logical OR of members of the enumeration `irtc_osc_cap_load_t`

status_t `IRTC_SetWriteProtection(RTC_Type *base, bool lock)`

Locks or unlocks IRTC registers for write access.

Note: When the registers are unlocked, they remain in unlocked state for 2 seconds, after which they are locked automatically. After power-on-reset, the registers come out unlocked and they are locked automatically 15 seconds after power on.

Parameters

- base – IRTC peripheral base address
- lock – true: Lock IRTC registers; false: Unlock IRTC registers.

Returns

kStatus_Success: if lock or unlock operation is successful kStatus_Fail: if lock or unlock operation fails even after multiple retry attempts

static inline void IRTC_Reset(RTC_Type *base)

Performs a software reset on the IRTC module.

Clears contents of alarm, interrupt (status and enable except tamper interrupt enable bit) registers, STATUS[*CMP_DONE*] and STATUS[*BUS_ERR*]. This has no effect on DST, calendar-ing, standby time and tamper detect registers.

Parameters

- base – IRTC peripheral base address

static inline void IRTC_Enable32kClkDuringRegisterWrite(RTC_Type *base, bool enable)

Enable/disable 32 kHz RTC OSC clock during RTC register write.

Parameters

- base – IRTC peripheral base address
- enable – Enable/disable 32 kHz RTC OSC clock.
 - true: Enables the oscillator.
 - false: Disables the oscillator.

void IRTC_ConfigClockOut(RTC_Type *base, *irtc_clockout_sel_t* clkOut)

Select which clock to output from RTC.

Select which clock to output from RTC for other modules to use inside SoC, for example, RTC subsystem needs RTC to output 1HZ clock for sub-second counter.

Parameters

- base – IRTC peripheral base address
- clkOut – select clock to use for output,

static inline uint8_t IRTC_GetTamperStatusFlag(RTC_Type *base)

Gets the IRTC Tamper status flags.

Parameters

- base – IRTC peripheral base address

Returns

The Tamper status value.

static inline void IRTC_ClearTamperStatusFlag(RTC_Type *base)

Gets the IRTC Tamper status flags.

Parameters

- base – IRTC peripheral base address

static inline void IRTC_SetTamperConfigurationOver(RTC_Type *base)

Set tamper configuration over.

Note that this API is needed after call IRTC_SetTamperParams to configure tamper events to notify IRTC module that tamper configuration process is over.

Parameters

- base – IRTC peripheral base address

IRTC_STATUS_W1C_BITS

struct __irtc_datetime

#include <fsl_irtc.h> Structure is used to hold the date and time.

Public Members

uint16_t year

Range from 1984 to 2239.

uint8_t month

Range from 1 to 12.

uint8_t day

Range from 1 to 31 (depending on month).

uint8_t weekDay

Range from 0(Sunday) to 6(Saturday).

uint8_t hour

Range from 0 to 23.

uint8_t minute

Range from 0 to 59.

uint8_t second

Range from 0 to 59.

struct __irtc_daylight_time

#include <fsl_irtc.h> Structure is used to hold the daylight saving time.

Public Members

uint8_t startMonth

Range from 1 to 12

uint8_t endMonth

Range from 1 to 12

uint8_t startDay

Range from 1 to 31 (depending on month)

uint8_t endDay

Range from 1 to 31 (depending on month)

uint8_t startHour

Range from 0 to 23

uint8_t endHour

Range from 0 to 23

struct __irtc_tamper_config

#include <fsl_irtc.h> Structure is used to define the parameters to configure a RTC tamper event.

Public Members

bool activePassive
true: configure tamper as active; false: passive tamper

bool direction
true: configure tamper direction as output; false: configure as input; this is only used if a tamper pin is defined as active

bool pinPolarity
true: tamper has active low polarity; false: active high polarity

irtc_filter_clock_source_t filterClk
Clock source for the tamper filter

uint8_t filterDuration
Tamper filter duration.

struct _irtc_config

#include <fsl_irtc.h> RTC config structure.

This structure holds the configuration settings for the RTC peripheral. To initialize this structure to reasonable defaults, call the `IRTC_GetDefaultConfig()` function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

Public Members

bool wakeupSelect
true: Tamper pin 0 is used to wakeup the chip; false: Tamper pin 0 is used as the tamper pin

bool timerStdMask
true: Sampling clocks gated in standby mode; false: Sampling clocks not gated

irtc_alarm_match_t alrmMatch
Pick one option from enumeration :: `irtc_alarm_match_t`

2.24 Common Driver

`FSL_COMMON_DRIVER_VERSION`

common driver version.

`DEBUG_CONSOLE_DEVICE_TYPE_NONE`

No debug console.

`DEBUG_CONSOLE_DEVICE_TYPE_UART`

Debug console based on UART.

`DEBUG_CONSOLE_DEVICE_TYPE_LPUART`

Debug console based on LPUART.

`DEBUG_CONSOLE_DEVICE_TYPE_LPSCI`

Debug console based on LPSCI.

`DEBUG_CONSOLE_DEVICE_TYPE_USBCDC`

Debug console based on USBCDC.

DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM

Debug console based on FLEXCOMM.

DEBUG_CONSOLE_DEVICE_TYPE_IUART

Debug console based on i.MX UART.

DEBUG_CONSOLE_DEVICE_TYPE_VUSART

Debug console based on LPC_VUSART.

DEBUG_CONSOLE_DEVICE_TYPE_MINI_UART

Debug console based on LPC_USART.

DEBUG_CONSOLE_DEVICE_TYPE_SWO

Debug console based on SWO.

DEBUG_CONSOLE_DEVICE_TYPE_QSCI

Debug console based on QSCI.

MIN(*a*, *b*)

Computes the minimum of *a* and *b*.

MAX(*a*, *b*)

Computes the maximum of *a* and *b*.

UINT16_MAX

Max value of uint16_t type.

UINT32_MAX

Max value of uint32_t type.

SDK_ATOMIC_LOCAL_ADD(*addr*, *val*)

Add value *val* from the variable at address *address*.

SDK_ATOMIC_LOCAL_SUB(*addr*, *val*)

Subtract value *val* to the variable at address *address*.

SDK_ATOMIC_LOCAL_SET(*addr*, *bits*)

Set the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR(*addr*, *bits*)

Clear the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_TOGGLE(*addr*, *bits*)

Toggle the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR_AND_SET(*addr*, *clearBits*, *setBits*)

For the variable at address *address*, clear the bits specified by *clearBits* and set the bits specified by *setBits*.

SDK_ATOMIC_LOCAL_COMPARE_AND_SET(*addr*, *expected*, *newValue*)

For the variable at address *address*, check whether the value equal to *expected*. If value same as *expected* then update *newValue* to address and return **true** , else return **false** .

SDK_ATOMIC_LOCAL_TEST_AND_SET(*addr*, *newValue*)

For the variable at address *address*, set as *newValue* value and return old value.

USEC_TO_COUNT(*us*, *clockFreqInHz*)

Macro to convert a microsecond period to raw count value

COUNT_TO_USEC(*count*, *clockFreqInHz*)

Macro to convert a raw count value to microsecond

MSEC_TO_COUNT(ms, clockFreqInHz)

Macro to convert a millisecond period to raw count value

COUNT_TO_MSEC(count, clockFreqInHz)

Macro to convert a raw count value to millisecond

SDK_ISR_EXIT_BARRIER

SDK_ALIGN(var, alignbytes)

Macro to define a variable with alignbytes alignment

SDK_SIZEALIGN(var, alignbytes)

Macro to define a variable with L1 d-cache line size alignment

Macro to define a variable with L2 cache line size alignment

Macro to change a value to a given size aligned value (rounded up)

SDK_SIZEALIGN_UP(var, alignbytes)

Macro to change a value to a given size aligned value (rounded up), the wrapper of SDK_SIZEALIGN

SDK_SIZEALIGN_DOWN(var, alignbytes)

Macro to change a value to a given size aligned value (rounded down)

SDK_IS_ALIGNED(var, alignbytes)

Macro to check if a value is aligned to a given size

AT_NONCACHEABLE_SECTION(var)

Define a variable *var*, and place it in non-cacheable section.

AT_NONCACHEABLE_SECTION_ALIGN(var, alignbytes)

Define a variable *var*, and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes*.

AT_NONCACHEABLE_SECTION_INIT(var)

Define a variable *var* with initial value, and place it in non-cacheable section.

AT_NONCACHEABLE_SECTION_ALIGN_INIT(var, alignbytes)

Define a variable *var* with initial value, and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes*.

AT_CACHE_LINE_SECTION(var)

Define a variable *var*, which is cache line size aligned and be placed in CacheLineData section.

AT_CACHE_LINE_SECTION_INIT(var)

Define a variable *var* with initial value, which is cache line size aligned and be placed in CacheLineData.init section.

AT_QUICKACCESS_SECTION_CODE(func)

Place function in a section which can be accessed quickly by core.

AT_QUICKACCESS_SECTION_DATA(var)

Place data in a section which can be accessed quickly by core.

AT_QUICKACCESS_SECTION_DATA_ALIGN(var, alignbytes)

Place data in a section which can be accessed quickly by core, and the variable address is set to align with *alignbytes*.

MCUX_RAMFUNC

Function attribute to place function in RAM. For example, to place function `my_func` in ram, use like:

```
MCUX_RAMFUNC my_func
```

RAMFUNCTION_SECTION_CODE(func)

Place function in ram.

enum __status_groups

Status group numbers.

Values:

enumerator `kStatusGroup_Generic`

Group number for generic status codes.

enumerator `kStatusGroup_FLASH`

Group number for FLASH status codes.

enumerator `kStatusGroup_LPSPI`

Group number for LPSPI status codes.

enumerator `kStatusGroup_FLEXIO_SPI`

Group number for FLEXIO SPI status codes.

enumerator `kStatusGroup_DSPI`

Group number for DSPI status codes.

enumerator `kStatusGroup_FLEXIO_UART`

Group number for FLEXIO UART status codes.

enumerator `kStatusGroup_FLEXIO_I2C`

Group number for FLEXIO I2C status codes.

enumerator `kStatusGroup_LPI2C`

Group number for LPI2C status codes.

enumerator `kStatusGroup_UART`

Group number for UART status codes.

enumerator `kStatusGroup_I2C`

Group number for UART status codes.

enumerator `kStatusGroup_LPSCI`

Group number for LPSCI status codes.

enumerator `kStatusGroup_LPUART`

Group number for LPUART status codes.

enumerator `kStatusGroup_SPI`

Group number for SPI status code.

enumerator `kStatusGroup_XRDC`

Group number for XRDC status code.

enumerator `kStatusGroup_SEMA42`

Group number for SEMA42 status code.

enumerator `kStatusGroup_SDHC`

Group number for SDHC status code

enumerator kStatusGroup_SDMMC
Group number for SDMMC status code

enumerator kStatusGroup_SAI
Group number for SAI status code

enumerator kStatusGroup_MCG
Group number for MCG status codes.

enumerator kStatusGroup_SCG
Group number for SCG status codes.

enumerator kStatusGroup_SDSPI
Group number for SDSPI status codes.

enumerator kStatusGroup_FLEXIO_I2S
Group number for FLEXIO I2S status codes

enumerator kStatusGroup_FLEXIO_MCULCD
Group number for FLEXIO LCD status codes

enumerator kStatusGroup_FLASHIAP
Group number for FLASHIAP status codes

enumerator kStatusGroup_FLEXCOMM_I2C
Group number for FLEXCOMM I2C status codes

enumerator kStatusGroup_I2S
Group number for I2S status codes

enumerator kStatusGroup_IUART
Group number for IUART status codes

enumerator kStatusGroup_CSI
Group number for CSI status codes

enumerator kStatusGroup_MIPI_DSI
Group number for MIPI DSI status codes

enumerator kStatusGroup_SDRAMC
Group number for SDRAMC status codes.

enumerator kStatusGroup_POWER
Group number for POWER status codes.

enumerator kStatusGroup_ENET
Group number for ENET status codes.

enumerator kStatusGroup_PHY
Group number for PHY status codes.

enumerator kStatusGroup_TRGMUX
Group number for TRGMUX status codes.

enumerator kStatusGroup_SMARTCARD
Group number for SMARTCARD status codes.

enumerator kStatusGroup_LMEM
Group number for LMEM status codes.

enumerator kStatusGroup_QSPI
Group number for QSPI status codes.

enumerator kStatusGroup_DMA
Group number for DMA status codes.

enumerator kStatusGroup_EDMA
Group number for EDMA status codes.

enumerator kStatusGroup_DMAMGR
Group number for DMAMGR status codes.

enumerator kStatusGroup_FLEXCAN
Group number for FlexCAN status codes.

enumerator kStatusGroup_LTC
Group number for LTC status codes.

enumerator kStatusGroup_FLEXIO_CAMERA
Group number for FLEXIO CAMERA status codes.

enumerator kStatusGroup_LPC_SPI
Group number for LPC_SPI status codes.

enumerator kStatusGroup_LPC_USART
Group number for LPC_USART status codes.

enumerator kStatusGroup_DMIC
Group number for DMIC status codes.

enumerator kStatusGroup_SDIF
Group number for SDIF status codes.

enumerator kStatusGroup_SPIFI
Group number for SPIFI status codes.

enumerator kStatusGroup_OTP
Group number for OTP status codes.

enumerator kStatusGroup_MCAN
Group number for MCAN status codes.

enumerator kStatusGroup_CAAM
Group number for CAAM status codes.

enumerator kStatusGroup_ECSPI
Group number for ECSPI status codes.

enumerator kStatusGroup_USDHC
Group number for USDHC status codes.

enumerator kStatusGroup_LPC_I2C
Group number for LPC_I2C status codes.

enumerator kStatusGroup_DCP
Group number for DCP status codes.

enumerator kStatusGroup_MSCAN
Group number for MSCAN status codes.

enumerator kStatusGroup_ESAI
Group number for ESAI status codes.

enumerator kStatusGroup_FLEXSPI
Group number for FLEXSPI status codes.

enumerator kStatusGroup_MMDC
Group number for MMDC status codes.

enumerator kStatusGroup_PDM
Group number for MIC status codes.

enumerator kStatusGroup_SDMA
Group number for SDMA status codes.

enumerator kStatusGroup_ICS
Group number for ICS status codes.

enumerator kStatusGroup_SPDIF
Group number for SPDIF status codes.

enumerator kStatusGroup_LPC_MINISPI
Group number for LPC_MINISPI status codes.

enumerator kStatusGroup_HASHCRYPT
Group number for Hashcrypt status codes

enumerator kStatusGroup_LPC_SPI_SSP
Group number for LPC_SPI_SSP status codes.

enumerator kStatusGroup_I3C
Group number for I3C status codes

enumerator kStatusGroup_LPC_I2C_1
Group number for LPC_I2C_1 status codes.

enumerator kStatusGroup_NOTIFIER
Group number for NOTIFIER status codes.

enumerator kStatusGroup_DebugConsole
Group number for debug console status codes.

enumerator kStatusGroup_SEMC
Group number for SEMC status codes.

enumerator kStatusGroup_ApplicationRangeStart
Starting number for application groups.

enumerator kStatusGroup_IAP
Group number for IAP status codes

enumerator kStatusGroup_SFA
Group number for SFA status codes

enumerator kStatusGroup_SPC
Group number for SPC status codes.

enumerator kStatusGroup_PUF
Group number for PUF status codes.

enumerator kStatusGroup_TOUCH_PANEL
Group number for touch panel status codes

enumerator kStatusGroup_VBAT
Group number for VBAT status codes

enumerator kStatusGroup_XSPI
Group number for XSPI status codes

enumerator kStatusGroup_PNGDEC
Group number for PNGDEC status codes

enumerator kStatusGroup_JPEGDEC
Group number for JPEGDEC status codes

enumerator kStatusGroup_AUDMIX
Group number for AUDMIX status codes

enumerator kStatusGroup_HAL_GPIO
Group number for HAL GPIO status codes.

enumerator kStatusGroup_HAL_UART
Group number for HAL UART status codes.

enumerator kStatusGroup_HAL_TIMER
Group number for HAL TIMER status codes.

enumerator kStatusGroup_HAL_SPI
Group number for HAL SPI status codes.

enumerator kStatusGroup_HAL_I2C
Group number for HAL I2C status codes.

enumerator kStatusGroup_HAL_FLASH
Group number for HAL FLASH status codes.

enumerator kStatusGroup_HAL_PWM
Group number for HAL PWM status codes.

enumerator kStatusGroup_HAL_RNG
Group number for HAL RNG status codes.

enumerator kStatusGroup_HAL_I2S
Group number for HAL I2S status codes.

enumerator kStatusGroup_HAL_ADC_SENSOR
Group number for HAL ADC SENSOR status codes.

enumerator kStatusGroup_TIMERMANAGER
Group number for TiMER MANAGER status codes.

enumerator kStatusGroup_SERIALMANAGER
Group number for SERIAL MANAGER status codes.

enumerator kStatusGroup_LED
Group number for LED status codes.

enumerator kStatusGroup_BUTTON
Group number for BUTTON status codes.

enumerator kStatusGroup_EXTERN_EEPROM
Group number for EXTERN EEPROM status codes.

enumerator kStatusGroup_SHELL
Group number for SHELL status codes.

enumerator kStatusGroup_MEM_MANAGER
Group number for MEM MANAGER status codes.

enumerator kStatusGroup_LIST
Group number for List status codes.

enumerator `kStatusGroup_OSA`
Group number for OSA status codes.

enumerator `kStatusGroup_COMMON_TASK`
Group number for Common task status codes.

enumerator `kStatusGroup_MSG`
Group number for messaging status codes.

enumerator `kStatusGroup_SDK_OCOTP`
Group number for OCOTP status codes.

enumerator `kStatusGroup_SDK_FLEXSPINOR`
Group number for FLEXSPINOR status codes.

enumerator `kStatusGroup_CODEEC`
Group number for codec status codes.

enumerator `kStatusGroup_ASRC`
Group number for codec status ASRC.

enumerator `kStatusGroup_OTFAD`
Group number for codec status codes.

enumerator `kStatusGroup_SDIOSLV`
Group number for SDIOSLV status codes.

enumerator `kStatusGroup_MECC`
Group number for MECC status codes.

enumerator `kStatusGroup_ENET_QOS`
Group number for ENET_QOS status codes.

enumerator `kStatusGroup_LOG`
Group number for LOG status codes.

enumerator `kStatusGroup_I3CBUS`
Group number for I3CBUS status codes.

enumerator `kStatusGroup_QSCI`
Group number for QSCI status codes.

enumerator `kStatusGroup_ELEMU`
Group number for ELEMU status codes.

enumerator `kStatusGroup_QUEUEDSPI`
Group number for QSPI status codes.

enumerator `kStatusGroup_POWER_MANAGER`
Group number for POWER_MANAGER status codes.

enumerator `kStatusGroup_IPED`
Group number for IPED status codes.

enumerator `kStatusGroup_ELS_PKC`
Group number for ELS PKC status codes.

enumerator `kStatusGroup_CSS_PKC`
Group number for CSS PKC status codes.

enumerator `kStatusGroup_HOSTIF`
Group number for HOSTIF status codes.

enumerator kStatusGroup_CLIF

Group number for CLIF status codes.

enumerator kStatusGroup_BMA

Group number for BMA status codes.

enumerator kStatusGroup_NETC

Group number for NETC status codes.

enumerator kStatusGroup_ELE

Group number for ELE status codes.

enumerator kStatusGroup_GLIKEY

Group number for GLIKEY status codes.

enumerator kStatusGroup_AON_POWER

Group number for AON_POWER status codes.

enumerator kStatusGroup_AON_COMMON

Group number for AON_COMMON status codes.

enumerator kStatusGroup_ENDAT3

Group number for ENDAT3 status codes.

enumerator kStatusGroup_HIPERFACE

Group number for HIPERFACE status codes.

enumerator kStatusGroup_NPX

Group number for NPX status codes.

enumerator kStatusGroup_ELA_CSEC

Group number for ELA_CSEC status codes.

enumerator kStatusGroup_FLEXIO_T_FORMAT

Group number for T-format status codes.

enumerator kStatusGroup_FLEXIO_A_FORMAT

Group number for A-format status codes.

enumerator kStatusGroup_LPC_QSPI

Group number for LPC QSPI status codes.

Generic status return codes.

Values:

enumerator kStatus_Success

Generic status for Success.

enumerator kStatus_Fail

Generic status for Fail.

enumerator kStatus_ReadOnly

Generic status for read only failure.

enumerator kStatus_OutOfRange

Generic status for out of range access.

enumerator kStatus_InvalidArgument

Generic status for invalid argument check.

enumerator kStatus_Timeout

Generic status for timeout.

enumerator kStatus_NoTransferInProgress

Generic status for no transfer in progress.

enumerator kStatus_Busy

Generic status for module is busy.

enumerator kStatus_NoData

Generic status for no data is found for the operation.

typedef int32_t status_t

Type used for all status and error return values.

void *SDK_Malloc(size_t size, size_t alignbytes)

Allocate memory with given alignment and aligned size.

This is provided to support the dynamically allocated memory used in cache-able region.

Parameters

- size – The length required to malloc.
- alignbytes – The alignment size.

Return values

The – allocated memory.

void SDK_Free(void *ptr)

Free memory.

Parameters

- ptr – The memory to be release.

void SDK_DelayAtLeastUs(uint32_t delayTime_us, uint32_t coreClock_Hz)

Delay at least for some time. Please note that, this API uses while loop for delay, different run-time environments make the time not precise, if precise delay count was needed, please implement a new delay function with hardware timer.

Parameters

- delayTime_us – Delay time in unit of microsecond.
- coreClock_Hz – Core clock frequency with Hz.

static inline status_t EnableIRQ(IRQn_Type interrupt)

Enable specific interrupt.

Enable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only enables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ number.

Return values

- kStatus_Success – Interrupt enabled successfully
- kStatus_Fail – Failed to enable the interrupt

static inline *status_t* DisableIRQ(IRQn_Type interrupt)

Disable specific interrupt.

Disable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only disables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ number.

Return values

- kStatus_Success – Interrupt disabled successfully
- kStatus_Fail – Failed to disable the interrupt

static inline *status_t* EnableIRQWithPriority(IRQn_Type interrupt, uint8_t priNum)

Enable the IRQ, and also set the interrupt priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ to Enable.
- priNum – Priority number set to interrupt controller register.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline *status_t* IRQ_SetPriority(IRQn_Type interrupt, uint8_t priNum)

Set the IRQ priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The IRQ to set.
- priNum – Priority number set to interrupt controller register.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline *status_t* IRQ_ClearPendingIRQ(IRQn_Type interrupt)

Clear the pending IRQ flag.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS.

Parameters

- interrupt – The flag which IRQ to clear.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline uint32_t DisableGlobalIRQ(void)

Disable the global IRQ.

Disable the global interrupt and return the current primask register. User is required to provided the primask register for the EnableGlobalIRQ().

Returns

Current primask value.

static inline void EnableGlobalIRQ(uint32_t primask)

Enable the global IRQ.

Set the primask register with the provided primask value but not just enable the primask. The idea is for the convenience of integration of RTOS. some RTOS get its own management mechanism of primask. User is required to use the EnableGlobalIRQ() and DisableGlobalIRQ() in pair.

Parameters

- primask – value of primask register to be restored. The primask value is supposed to be provided by the DisableGlobalIRQ().

static inline bool _SDK_AtomicLocalCompareAndSet(uint32_t *addr, uint32_t expected, uint32_t
newValue)

static inline uint32_t _SDK_AtomicTestAndSet(uint32_t *addr, uint32_t newValue)

FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ

Macro to use the default weak IRQ handler in drivers.

MAKE_STATUS(group, code)

Construct a status code value from a group and code number.

MAKE_VERSION(major, minor, bugfix)

Construct the version number for drivers.

The driver version is a 32-bit number, for both 32-bit platforms(such as Cortex M) and 16-bit platforms(such as DSC).

Unused	Major Version	Minor Version	Bug Fix	
31	25 24	17 16	9 8	0

ARRAY_SIZE(x)

Computes the number of elements in an array.

UINT64_H(X)

Macro to get upper 32 bits of a 64-bit value

UINT64_L(X)

Macro to get lower 32 bits of a 64-bit value

SUPPRESS_FALL_THROUGH_WARNING()

For switch case code block, if case section ends without “break;” statement, there will be fallthrough warning with compiler flag -Wextra or -Wimplicit-fallthrough=n when using armgcc. To suppress this warning, “SUPPRESS_FALL_THROUGH_WARNING();” need to be added at the end of each case section which misses “break;”statement.

MSDK_REG_SECURE_ADDR(x)

Convert the register address to the one used in secure mode.

MSDK_REG_NONSECURE_ADDR(x)

Convert the register address to the one used in non-secure mode.

MSDK_HAS_DWT_CYCCNT

The chip supports DWT CYCCNT or not.

MSDK_INVALID_IRQ_HANDLER

Invalid IRQ handler address.

2.25 Lin_lpuart_driver

FSL_LIN_LPUART_DRIVER_VERSION

LIN LPUART driver version.

enum _lin_lpuart_stop_bit_count

Values:

enumerator kLPUART_OneStopBit

One stop bit

enumerator kLPUART_TwoStopBit

Two stop bits

enum _lin_lpuart_flags

Values:

enumerator kLPUART_TxDataRegEmptyFlag

Transmit data register empty flag, sets when transmit buffer is empty

enumerator kLPUART_TransmissionCompleteFlag

Transmission complete flag, sets when transmission activity complete

enumerator kLPUART_RxDataRegFullFlag

Receive data register full flag, sets when the receive data buffer is full

enumerator kLPUART_IdleLineFlag

Idle line detect flag, sets when idle line detected

enumerator kLPUART_RxOverrunFlag

Receive Overrun, sets when new data is received before data is read from receive register

enumerator kLPUART_NoiseErrorFlag

Receive takes 3 samples of each received bit. If any of these samples differ, noise flag sets

enumerator kLPUART_FramingErrorFlag

Frame error flag, sets if logic 0 was detected where stop bit expected

enumerator kLPUART_ParityErrorFlag

If parity enabled, sets upon parity error detection

enumerator kLPUART_LinBreakFlag

LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled

enumerator kLPUART_RxActiveEdgeFlag

Receive pin active edge interrupt flag, sets when active edge detected

enumerator kLPUART_RxActiveFlag

Receiver Active Flag (RAF), sets at beginning of valid start bit

enumerator kLPUART_DataMatch1Flag

The next character to be read from LPUART_DATA matches MA1

enumerator kLPUART_DataMatch2Flag

The next character to be read from LPUART_DATA matches MA2

enumerator kLPUART_NoiseErrorInRxDataRegFlag

NOISY bit, sets if noise detected in current data word

enumerator kLPUART_ParityErrorInRxDataRegFlag

PARITY bit, sets if noise detected in current data word

enumerator kLPUART_TxFifoEmptyFlag

TXEMPT bit, sets if transmit buffer is empty

enumerator kLPUART_RxFifoEmptyFlag

RXEMPT bit, sets if receive buffer is empty

enumerator kLPUART_TxFifoOverflowFlag

TXOF bit, sets if transmit buffer overflow occurred

enumerator kLPUART_RxFifoUnderflowFlag

RXUF bit, sets if receive buffer underflow occurred

enum _lin_lpuart_interrupt_enable

Values:

enumerator kLPUART_LinBreakInterruptEnable

LIN break detect.

enumerator kLPUART_RxActiveEdgeInterruptEnable

Receive Active Edge.

enumerator kLPUART_TxDataRegEmptyInterruptEnable

Transmit data register empty.

enumerator kLPUART_TransmissionCompleteInterruptEnable

Transmission complete.

enumerator kLPUART_RxDataRegFullInterruptEnable

Receiver data register full.

enumerator kLPUART_IdleLineInterruptEnable

Idle line.

enumerator kLPUART_RxOverrunInterruptEnable

Receiver Overrun.

enumerator kLPUART_NoiseErrorInterruptEnable

Noise error flag.

enumerator kLPUART_FramingErrorInterruptEnable

Framing error flag.

enumerator kLPUART_ParityErrorInterruptEnable

Parity error flag.

enumerator kLPUART_TxFifoOverflowInterruptEnable

Transmit FIFO Overflow.

enumerator kLPUART_RxFifoUnderflowInterruptEnable

Receive FIFO Underflow.

enum _lin_lpuart_status

Values:

enumerator kStatus_LPUART_TxBusy

TX busy

enumerator kStatus_LPUART_RxBusy

RX busy

enumerator kStatus_LPUART_TxIdle

LPUART transmitter is idle.

enumerator kStatus_LPUART_RxIdle

LPUART receiver is idle.

enumerator kStatus_LPUART_TxWatermarkTooLarge

TX FIFO watermark too large

enumerator kStatus_LPUART_RxWatermarkTooLarge

RX FIFO watermark too large

enumerator kStatus_LPUART_FlagCannotClearManually

Some flag can't manually clear

enumerator kStatus_LPUART_Error

Error happens on LPUART.

enumerator kStatus_LPUART_RxRingBufferOverrun

LPUART RX software ring buffer overrun.

enumerator kStatus_LPUART_RxHardwareOverrun

LPUART RX receiver overrun.

enumerator kStatus_LPUART_NoiseError

LPUART noise error.

enumerator kStatus_LPUART_FramingError

LPUART framing error.

enumerator kStatus_LPUART_ParityError

LPUART parity error.

enum lin_lpuart_bit_count_per_char_t

Values:

enumerator LPUART_8_BITS_PER_CHAR

8-bit data characters

enumerator LPUART_9_BITS_PER_CHAR

9-bit data characters

enumerator LPUART_10_BITS_PER_CHAR

10-bit data characters

typedef enum *lin_lpuart_stop_bit_count* lin_lpuart_stop_bit_count_t

static inline bool LIN_LPUART_GetRxDataPolarity(const LPUART_Type *base)

static inline void LIN_LPUART_SetRxDataPolarity(LPUART_Type *base, bool polarity)

static inline void LIN_LPUART_WriteByte(LPUART_Type *base, uint8_t data)

static inline void LIN_LPUART_ReadByte(const LPUART_Type *base, uint8_t *readData)

status_t LIN_LPUART_CalculateBaudRate(LPUART_Type *base, uint32_t baudRate_Bps,
uint32_t srcClock_Hz, uint32_t *osr, uint16_t *sbr)

Calculates the best osr and sbr value for configured baudrate.

Parameters

- base – LPUART peripheral base address
- baudRate_Bps – user configuration structure of type #lin_user_config_t
- srcClock_Hz – pointer to the LIN_LPUART driver state structure
- osr – pointer to osr value
- sbr – pointer to sbr value

Returns

An error code or lin_status_t

void LIN_LPUART_SetBaudRate(LPUART_Type *base, uint32_t *osr, uint16_t *sbr)

Configure baudrate according to osr and sbr value.

Parameters

- base – LPUART peripheral base address
- osr – pointer to osr value
- sbr – pointer to sbr value

lin_status_t LIN_LPUART_Init(LPUART_Type *base, lin_user_config_t *linUserConfig,
lin_state_t *linCurrentState, uint32_t linSourceClockFreq)

Initializes an LIN_LPUART instance for LIN Network.

The caller provides memory for the driver state structures during initialization. The user must select the LIN_LPUART clock source in the application to initialize the LIN_LPUART. This function initializes a LPUART instance for operation. This function will initialize the run-time state structure to keep track of the on-going transfers, initialize the module to user defined settings and default settings, set break field length to be 13 bit times minimum, enable the break detect interrupt, Rx complete interrupt, frame error detect interrupt, and enable the LPUART module transmitter and receiver

Parameters

- base – LPUART peripheral base address
- linUserConfig – user configuration structure of type #lin_user_config_t
- linCurrentState – pointer to the LIN_LPUART driver state structure
- linSourceClockFreq – LIN source clock frequency in Hz

Returns

An error code or `lin_status_t`

`lin_status_t` LIN_LPUART_Deinit(LPUART_Type *base)

Shuts down the LIN_LPUART by disabling interrupts and transmitter/receiver.

Parameters

- base – LPUART peripheral base address

Returns

An error code or `lin_status_t`

`lin_status_t` LIN_LPUART_SendFrameDataBlocking(LPUART_Type *base, const uint8_t *txBuff, uint8_t txSize, uint32_t timeoutMSec)

Sends Frame data out through the LIN_LPUART module using blocking method. This function will calculate the checksum byte and send it with the frame data. Blocking means that the function does not return until the transmission is complete.

Parameters

- base – LPUART peripheral base address
- txBuff – source buffer containing 8-bit data chars to send
- txSize – the number of bytes to send
- timeoutMSec – timeout value in milli seconds

Returns

An error code or `lin_status_t`

`lin_status_t` LIN_LPUART_SendFrameData(LPUART_Type *base, const uint8_t *txBuff, uint8_t txSize)

Sends frame data out through the LIN_LPUART module using non-blocking method. This enables an a-sync method for transmitting data. Non-blocking means that the function returns immediately. The application has to get the transmit status to know when the transmit is complete. This function will calculate the checksum byte and send it with the frame data.

Parameters

- base – LPUART peripheral base address
- txBuff – source buffer containing 8-bit data chars to send
- txSize – the number of bytes to send

Returns

An error code or `lin_status_t`

`lin_status_t` LIN_LPUART_GetTransmitStatus(LPUART_Type *base, uint8_t *bytesRemaining)

Get status of an on-going non-blocking transmission While sending frame data using non-blocking method, users can use this function to get status of that transmission. This function return LIN_TX_BUSY while sending, or LIN_TIMEOUT if timeout has occurred, or return LIN_SUCCESS when the transmission is complete. The bytesRemaining shows number of bytes that still needed to transmit.

Parameters

- base – LPUART peripheral base address
- bytesRemaining – Number of bytes still needed to transmit

Returns

`lin_status_t` LIN_TX_BUSY, LIN_SUCCESS or LIN_TIMEOUT

`lin_status_t LIN_LPUART_RecvFrmDataBlocking(LPUART_Type *base, uint8_t *rxBuff, uint8_t rxSize, uint32_t timeoutMSec)`

Receives frame data through the LIN_LPUART module using blocking method. This function will check the checksum byte. If the checksum is correct, it will receive the frame data. Blocking means that the function does not return until the reception is complete.

Parameters

- `base` – LPUART peripheral base address
- `rxBuff` – buffer containing 8-bit received data
- `rxSize` – the number of bytes to receive
- `timeoutMSec` – timeout value in milli seconds

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_RecvFrmData(LPUART_Type *base, uint8_t *rxBuff, uint8_t rxSize)`

Receives frame data through the LIN_LPUART module using non-blocking method. This function will check the checksum byte. If the checksum is correct, it will receive it with the frame data. Non-blocking means that the function returns immediately. The application has to get the receive status to know when the reception is complete.

Parameters

- `base` – LPUART peripheral base address
- `rxBuff` – buffer containing 8-bit received data
- `rxSize` – the number of bytes to receive

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_AbortTransferData(LPUART_Type *base)`

Aborts an on-going non-blocking transmission/reception. While performing a non-blocking transferring data, users can call this function to terminate immediately the transferring.

Parameters

- `base` – LPUART peripheral base address

Returns

An error code or `lin_status_t`

`lin_status_t LIN_LPUART_GetReceiveStatus(LPUART_Type *base, uint8_t *bytesRemaining)`

Get status of an on-going non-blocking reception. While receiving frame data using non-blocking method, users can use this function to get status of that receiving. This function returns the current event ID, `LIN_RX_BUSY` while receiving and return `LIN_SUCCESS`, or timeout (`LIN_TIMEOUT`) when the reception is complete. The `bytesRemaining` shows number of bytes that still needed to receive.

Parameters

- `base` – LPUART peripheral base address
- `bytesRemaining` – Number of bytes still needed to receive

Returns

`lin_status_t` `LIN_RX_BUSY`, `LIN_TIMEOUT` or `LIN_SUCCESS`

`lin_status_t LIN_LPUART_GoToSleepMode(LPUART_Type *base)`

This function puts current node to sleep mode. This function changes current node state to `LIN_NODE_STATE_SLEEP_MODE`.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_GotoIdleState(LPUART_Type *base)

Puts current LIN node to Idle state This function changes current node state to LIN_NODE_STATE_IDLE.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_SendWakeupSignal(LPUART_Type *base)

Sends a wakeup signal through the LIN_LPUART interface.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_MasterSendHeader(LPUART_Type *base, uint8_t id)

Sends frame header out through the LIN_LPUART module using a non-blocking method. This function sends LIN Break field, sync field then the ID with correct parity.

Parameters

- base – LPUART peripheral base address
- id – Frame Identifier

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_EnableIRQ(LPUART_Type *base)

Enables LIN_LPUART hardware interrupts.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_DisableIRQ(LPUART_Type *base)

Disables LIN_LPUART hardware interrupts.

Parameters

- base – LPUART peripheral base address

Returns

An error code or lin_status_t

lin_status_t LIN_LPUART_AutoBaudCapture(uint32_t instance)

This function capture bits time to detect break char, calculate baudrate from sync bits and enable transceiver if autobaud successful. This function should only be used in Slave. The timer should be in mode input capture of both rising and falling edges. The timer input capture pin should be externally connected to RXD pin.

Parameters

- instance – LPUART instance

Returns

lin_status_t

void LIN_LPUART_IRQHandler(LPUART_Type *base)

LIN_LPUART RX TX interrupt handler.

Parameters

- base – LPUART peripheral base address

LIN_LPUART_TRANSMISSION_COMPLETE_TIMEOUT

Max loops to wait for LPUART transmission complete.

When de-initializing the LIN LPUART module, the program shall wait for the previous transmission to complete. This parameter defines how many loops to check completion before return error. If defined as 0, driver will wait forever until completion.

AUTOBAUD_BAUDRATE_TOLERANCE

BIT_RATE_TOLERANCE_UNSYNC

BIT_DURATION_MAX_19200

BIT_DURATION_MIN_19200

BIT_DURATION_MAX_14400

BIT_DURATION_MIN_14400

BIT_DURATION_MAX_9600

BIT_DURATION_MIN_9600

BIT_DURATION_MAX_4800

BIT_DURATION_MIN_4800

BIT_DURATION_MAX_2400

BIT_DURATION_MIN_2400

TWO_BIT_DURATION_MAX_19200

TWO_BIT_DURATION_MIN_19200

TWO_BIT_DURATION_MAX_14400

TWO_BIT_DURATION_MIN_14400

TWO_BIT_DURATION_MAX_9600

TWO_BIT_DURATION_MIN_9600

TWO_BIT_DURATION_MAX_4800

TWO_BIT_DURATION_MIN_4800

TWO_BIT_DURATION_MAX_2400

TWO_BIT_DURATION_MIN_2400

AUTOBAUD_BREAK_TIME_MIN

2.26 LLWU: Low-Leakage Wakeup Unit Driver

```
static inline void LLWU_GetVersionId(LLWU_Type *base, llwu_version_id_t *versionId)
```

Gets the LLWU version ID.

This function gets the LLWU version ID, including the major version number, the minor version number, and the feature specification number.

Parameters

- `base` – LLWU peripheral base address.
- `versionId` – A pointer to the version ID structure.

```
static inline void LLWU_GetParam(LLWU_Type *base, llwu_param_t *param)
```

Gets the LLWU parameter.

This function gets the LLWU parameter, including a wakeup pin number, a module number, a DMA number, and a pin filter number.

Parameters

- `base` – LLWU peripheral base address.
- `param` – A pointer to the LLWU parameter structure.

```
void LLWU_SetExternalWakeupPinMode(LLWU_Type *base, uint32_t pinIndex,  
                                   llwu_external_pin_mode_t pinMode)
```

Sets the external input pin source mode.

This function sets the external input pin source mode that is used as a wake up source.

Parameters

- `base` – LLWU peripheral base address.
- `pinIndex` – A pin index to be enabled as an external wakeup source starting from 1.
- `pinMode` – A pin configuration mode defined in the `llwu_external_pin_modes_t`.

```
bool LLWU_GetExternalWakeupPinFlag(LLWU_Type *base, uint32_t pinIndex)
```

Gets the external wakeup source flag.

This function checks the external pin flag to detect whether the MCU is woken up by the specific pin.

Parameters

- `base` – LLWU peripheral base address.
- `pinIndex` – A pin index, which starts from 1.

Returns

True if the specific pin is a wakeup source.

```
void LLWU_ClearExternalWakeupPinFlag(LLWU_Type *base, uint32_t pinIndex)
```

Clears the external wakeup source flag.

This function clears the external wakeup source flag for a specific pin.

Parameters

- `base` – LLWU peripheral base address.
- `pinIndex` – A pin index, which starts from 1.

```
static inline void LLWU_EnableInternalModuleInterruptWakeup(LLWU_Type *base, uint32_t
                                                           moduleIndex, bool enable)
```

Enables/disables the internal module source.

This function enables/disables the internal module source mode that is used as a wake up source.

Parameters

- base – LLWU peripheral base address.
- moduleIndex – A module index to be enabled as an internal wakeup source starting from 1.
- enable – An enable or a disable setting

```
static inline void LLWU_EnableInternalModuleDmaRequestWakeup(LLWU_Type *base, uint32_t
                                                            moduleIndex, bool enable)
```

Enables/disables the internal module DMA wakeup source.

This function enables/disables the internal DMA that is used as a wake up source.

Parameters

- base – LLWU peripheral base address.
- moduleIndex – An internal module index which is used as a DMA request source, starting from 1.
- enable – Enable or disable the DMA request source

```
void LLWU_SetPinFilterMode(LLWU_Type *base, uint32_t filterIndex,
                          llwu_external_pin_filter_mode_t filterMode)
```

Sets the pin filter configuration.

This function sets the pin filter configuration.

Parameters

- base – LLWU peripheral base address.
- filterIndex – A pin filter index used to enable/disable the digital filter, starting from 1.
- filterMode – A filter mode configuration

```
bool LLWU_GetPinFilterFlag(LLWU_Type *base, uint32_t filterIndex)
```

Gets the pin filter configuration.

This function gets the pin filter flag.

Parameters

- base – LLWU peripheral base address.
- filterIndex – A pin filter index, which starts from 1.

Returns

True if the flag is a source of the existing low-leakage power mode.

```
void LLWU_ClearPinFilterFlag(LLWU_Type *base, uint32_t filterIndex)
```

Clears the pin filter configuration.

This function clears the pin filter flag.

Parameters

- base – LLWU peripheral base address.
- filterIndex – A pin filter index to clear the flag, starting from 1.

```
void LLWU_SetResetPinMode(LLWU_Type *base, bool pinEnable, bool pinFilterEnable)
```

Sets the reset pin mode.

This function determines how the reset pin is used as a low leakage mode exit source.

Parameters

- `base` – LLWU peripheral base address.
- `pinEnable` – Enable reset the pin filter
- `pinFilterEnable` – Specify whether the pin filter is enabled in Low-Leakage power mode.

```
FSL_LLWU_DRIVER_VERSION
```

LLWU driver version.

```
enum _llwu_external_pin_mode
```

External input pin control modes.

Values:

```
enumerator kLLWU_ExternalPinDisable
```

Pin disabled as a wakeup input.

```
enumerator kLLWU_ExternalPinRisingEdge
```

Pin enabled with the rising edge detection.

```
enumerator kLLWU_ExternalPinFallingEdge
```

Pin enabled with the falling edge detection.

```
enumerator kLLWU_ExternalPinAnyEdge
```

Pin enabled with any change detection.

```
enum _llwu_pin_filter_mode
```

Digital filter control modes.

Values:

```
enumerator kLLWU_PinFilterDisable
```

Filter disabled.

```
enumerator kLLWU_PinFilterRisingEdge
```

Filter positive edge detection.

```
enumerator kLLWU_PinFilterFallingEdge
```

Filter negative edge detection.

```
enumerator kLLWU_PinFilterAnyEdge
```

Filter any edge detection.

```
typedef enum _llwu_external_pin_mode llwu_external_pin_mode_t
```

External input pin control modes.

```
typedef enum _llwu_pin_filter_mode llwu_pin_filter_mode_t
```

Digital filter control modes.

```
typedef struct _llwu_version_id llwu_version_id_t
```

IP version ID definition.

```
typedef struct _llwu_param llwu_param_t
```

IP parameter definition.

```
typedef struct _llwu_external_pin_filter_mode llwu_external_pin_filter_mode_t
```

An external input pin filter control structure.

LLWU_REG_VAL(x)

struct __llwu_version_id
#include <fsl_llwu.h> IP version ID definition.

Public Members

uint16_t feature
A feature specification number.

uint8_t minor
The minor version number.

uint8_t major
The major version number.

struct __llwu_param
#include <fsl_llwu.h> IP parameter definition.

Public Members

uint8_t filters
A number of the pin filter.

uint8_t dmas
A number of the wakeup DMA.

uint8_t modules
A number of the wakeup module.

uint8_t pins
A number of the wake up pin.

struct __llwu_external_pin_filter_mode
#include <fsl_llwu.h> An external input pin filter control structure.

Public Members

uint32_t pinIndex
A pin number

llwu_pin_filter_mode_t filterMode
Filter mode

2.27 LPTMR: Low-Power Timer

void LPTMR_Init(LPTMR_Type *base, const *lptmr_config_t* *config)
Un gates the LPTMR clock and configures the peripheral for a basic operation.

Note: This API should be called at the beginning of the application using the LPTMR driver.

Parameters

- base – LPTMR peripheral base address

- config – A pointer to the LPTMR configuration structure.

void LPTMR_Deinit(LPTMR_Type *base)

Gates the LPTMR clock.

Parameters

- base – LPTMR peripheral base address

void LPTMR_GetDefaultConfig(*lptmr_config_t* *config)

Fills in the LPTMR configuration structure with default settings.

The default values are as follows.

```
config->timerMode = kLPTMR_TimerModeTimeCounter;
config->pinSelect = kLPTMR_PinSelectInput_0;
config->pinPolarity = kLPTMR_PinPolarityActiveHigh;
config->enableFreeRunning = false;
config->bypassPrescaler = true;
config->prescalerClockSource = kLPTMR_PrescalerClock_1;
config->value = kLPTMR_Prescale_Glitch_0;
```

Parameters

- config – A pointer to the LPTMR configuration structure.

static inline void LPTMR_EnableInterrupts(LPTMR_Type *base, uint32_t mask)

Enables the selected LPTMR interrupts.

Parameters

- base – LPTMR peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration `lptmr_interrupt_enable_t`

static inline void LPTMR_DisableInterrupts(LPTMR_Type *base, uint32_t mask)

Disables the selected LPTMR interrupts.

Parameters

- base – LPTMR peripheral base address
- mask – The interrupts to disable. This is a logical OR of members of the enumeration `lptmr_interrupt_enable_t`.

static inline uint32_t LPTMR_GetEnabledInterrupts(LPTMR_Type *base)

Gets the enabled LPTMR interrupts.

Parameters

- base – LPTMR peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `lptmr_interrupt_enable_t`

static inline uint32_t LPTMR_GetStatusFlags(LPTMR_Type *base)

Gets the LPTMR status flags.

Parameters

- base – LPTMR peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration `lptmr_status_flags_t`

```
static inline void LPTMR_ClearStatusFlags(LPTMR_Type *base, uint32_t mask)
```

Clears the LPTMR status flags.

Parameters

- base – LPTMR peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration `lptmr_status_flags_t`.

```
static inline void LPTMR_SetTimerPeriod(LPTMR_Type *base, uint32_t ticks)
```

Sets the timer period in units of count.

Timers counts from 0 until it equals the count value set here. The count value is written to the CMR register.

Note:

- a. The TCF flag is set with the CNR equals the count provided here and then increments.
 - b. Call the utility macros provided in the `fsl_common.h` to convert to ticks.
-

Parameters

- base – LPTMR peripheral base address
- ticks – A timer period in units of ticks

```
static inline uint32_t LPTMR_GetCurrentTimerCount(LPTMR_Type *base)
```

Reads the current timer counting value.

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note: Call the utility macros provided in the `fsl_common.h` to convert ticks to usec or msec.

Parameters

- base – LPTMR peripheral base address

Returns

The current counter value in ticks

```
static inline void LPTMR_StartTimer(LPTMR_Type *base)
```

Starts the timer.

After calling this function, the timer counts up to the CMR register value. Each time the timer reaches the CMR value and then increments, it generates a trigger pulse and sets the timeout interrupt flag. An interrupt is also triggered if the timer interrupt is enabled.

Parameters

- base – LPTMR peripheral base address

```
static inline void LPTMR_StopTimer(LPTMR_Type *base)
```

Stops the timer.

This function stops the timer and resets the timer's counter register.

Parameters

- base – LPTMR peripheral base address

```
FSL_LPTMR_DRIVER_VERSION
```

Driver Version

enum _lptmr_pin_select

LPTMR pin selection used in pulse counter mode.

Values:

enumerator kLPTMR_PinSelectInput_0

Pulse counter input 0 is selected

enumerator kLPTMR_PinSelectInput_1

Pulse counter input 1 is selected

enumerator kLPTMR_PinSelectInput_2

Pulse counter input 2 is selected

enumerator kLPTMR_PinSelectInput_3

Pulse counter input 3 is selected

enum _lptmr_pin_polarity

LPTMR pin polarity used in pulse counter mode.

Values:

enumerator kLPTMR_PinPolarityActiveHigh

Pulse Counter input source is active-high

enumerator kLPTMR_PinPolarityActiveLow

Pulse Counter input source is active-low

enum _lptmr_timer_mode

LPTMR timer mode selection.

Values:

enumerator kLPTMR_TimerModeTimeCounter

Time Counter mode

enumerator kLPTMR_TimerModePulseCounter

Pulse Counter mode

enum _lptmr_prescaler_glitch_value

LPTMR prescaler/glitch filter values.

Values:

enumerator kLPTMR_Prescale_Glitch_0

Prescaler divide 2, glitch filter does not support this setting

enumerator kLPTMR_Prescale_Glitch_1

Prescaler divide 4, glitch filter 2

enumerator kLPTMR_Prescale_Glitch_2

Prescaler divide 8, glitch filter 4

enumerator kLPTMR_Prescale_Glitch_3

Prescaler divide 16, glitch filter 8

enumerator kLPTMR_Prescale_Glitch_4

Prescaler divide 32, glitch filter 16

enumerator kLPTMR_Prescale_Glitch_5

Prescaler divide 64, glitch filter 32

enumerator kLPTMR_Prescale_Glitch_6

Prescaler divide 128, glitch filter 64

enumerator kLPTMR_Prescale_Glitch_7
Prescaler divide 256, glitch filter 128

enumerator kLPTMR_Prescale_Glitch_8
Prescaler divide 512, glitch filter 256

enumerator kLPTMR_Prescale_Glitch_9
Prescaler divide 1024, glitch filter 512

enumerator kLPTMR_Prescale_Glitch_10
Prescaler divide 2048 glitch filter 1024

enumerator kLPTMR_Prescale_Glitch_11
Prescaler divide 4096, glitch filter 2048

enumerator kLPTMR_Prescale_Glitch_12
Prescaler divide 8192, glitch filter 4096

enumerator kLPTMR_Prescale_Glitch_13
Prescaler divide 16384, glitch filter 8192

enumerator kLPTMR_Prescale_Glitch_14
Prescaler divide 32768, glitch filter 16384

enumerator kLPTMR_Prescale_Glitch_15
Prescaler divide 65536, glitch filter 32768

enum _lptmr_prescaler_clock_select
LPTMR prescaler/glitch filter clock select.

Note: Clock connections are SoC-specific

Values:

enumerator kLPTMR_PrescalerClock_0
Prescaler/glitch filter clock 0 selected.

enumerator kLPTMR_PrescalerClock_1
Prescaler/glitch filter clock 1 selected.

enumerator kLPTMR_PrescalerClock_2
Prescaler/glitch filter clock 2 selected.

enumerator kLPTMR_PrescalerClock_3
Prescaler/glitch filter clock 3 selected.

enum _lptmr_interrupt_enable
List of the LPTMR interrupts.

Values:

enumerator kLPTMR_TimerInterruptEnable
Timer interrupt enable

enum _lptmr_status_flags
List of the LPTMR status flags.

Values:

enumerator kLPTMR_TimerCompareFlag
Timer compare flag


```
typedef enum _lptmr_pin_select lptmr_pin_select_t
```

LPTMR pin selection used in pulse counter mode.

```
typedef enum _lptmr_pin_polarity lptmr_pin_polarity_t
```

LPTMR pin polarity used in pulse counter mode.

```
typedef enum _lptmr_timer_mode lptmr_timer_mode_t
```

LPTMR timer mode selection.

```
typedef enum _lptmr_prescaler_glitch_value lptmr_prescaler_glitch_value_t
```

LPTMR prescaler/glitch filter values.

```
typedef enum _lptmr_prescaler_clock_select lptmr_prescaler_clock_select_t
```

LPTMR prescaler/glitch filter clock select.

Note: Clock connections are SoC-specific

```
typedef enum _lptmr_interrupt_enable lptmr_interrupt_enable_t
```

List of the LPTMR interrupts.

```
typedef enum _lptmr_status_flags lptmr_status_flags_t
```

List of the LPTMR status flags.

```
typedef struct _lptmr_config lptmr_config_t
```

LPTMR config structure.

This structure holds the configuration settings for the LPTMR peripheral. To initialize this structure to reasonable defaults, call the `LPTMR_GetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration struct can be made constant so it resides in flash.

```
static inline void LPTMR_EnableTimerDMA(LPTMR_Type *base, bool enable)
```

Enable or disable timer DMA request.

Parameters

- `base` – base LPTMR peripheral base address
- `enable` – Switcher of timer DMA feature. “true” means to enable, “false” means to disable.

```
struct _lptmr_config
```

`#include <fsl_lptmr.h>` LPTMR config structure.

This structure holds the configuration settings for the LPTMR peripheral. To initialize this structure to reasonable defaults, call the `LPTMR_GetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration struct can be made constant so it resides in flash.

Public Members

```
lptmr_timer_mode_t timerMode
```

Time counter mode or pulse counter mode

```
lptmr_pin_select_t pinSelect
```

LPTMR pulse input pin select; used only in pulse counter mode

```
lptmr_pin_polarity_t pinPolarity
```

LPTMR pulse input pin polarity; used only in pulse counter mode

bool enableFreeRunning

True: enable free running, counter is reset on overflow False: counter is reset when the compare flag is set

bool bypassPrescaler

True: bypass prescaler; false: use clock from prescaler

lptmr_prescaler_clock_select_t prescalerClockSource

LPTMR clock source

lptmr_prescaler_glitch_value_t value

Prescaler or glitch filter value

2.28 LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver

2.29 LPUART DMA Driver

```
void LPUART__TransferCreateHandleDMA(LPUART_Type *base, lpuart_dma_handle_t *handle,
                                     lpuart_dma_transfer_callback_t callback, void
                                     *userData, dma_handle_t *txDmaHandle,
                                     dma_handle_t *rxDmaHandle)
```

Initializes the LPUART handle which is used in transactional functions.

Note: This function disables all LPUART interrupts.

Parameters

- base – LPUART peripheral base address.
- handle – Pointer to *lpuart_dma_handle_t* structure.
- callback – Callback function.
- userData – User data.
- txDmaHandle – User-requested DMA handle for TX DMA transfer.
- rxDmaHandle – User-requested DMA handle for RX DMA transfer.

```
status_t LPUART__TransferSendDMA(LPUART_Type *base, lpuart_dma_handle_t *handle,
                                 lpuart_transfer_t *xfer)
```

Sends data using DMA.

This function sends data using DMA. This is a non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- xfer – LPUART DMA transfer structure. See *lpuart_transfer_t*.

Return values

- kStatus__Success – if succeed, others failed.
- kStatus_LPUART_TxBusy – Previous transfer on going.

- `kStatus_InvalidArgument` – Invalid argument.

`status_t LPUART_TransferReceiveDMA(LPUART_Type *base, lpuart_dma_handle_t *handle, lpuart_transfer_t *xfer)`

Receives data using DMA.

This function receives data using DMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – Pointer to `lpuart_dma_handle_t` structure.
- `xfer` – LPUART DMA transfer structure. See `lpuart_transfer_t`.

Return values

- `kStatus_Success` – if succeed, others failed.
- `kStatus_LPUART_RxBusy` – Previous transfer on going.
- `kStatus_InvalidArgument` – Invalid argument.

`void LPUART_TransferAbortSendDMA(LPUART_Type *base, lpuart_dma_handle_t *handle)`

Aborts the sent data using DMA.

This function aborts send data using DMA.

Parameters

- `base` – LPUART peripheral base address
- `handle` – Pointer to `lpuart_dma_handle_t` structure

`void LPUART_TransferAbortReceiveDMA(LPUART_Type *base, lpuart_dma_handle_t *handle)`

Aborts the received data using DMA.

This function aborts the received data using DMA.

Parameters

- `base` – LPUART peripheral base address
- `handle` – Pointer to `lpuart_dma_handle_t` structure

`status_t LPUART_TransferGetSendCountDMA(LPUART_Type *base, lpuart_dma_handle_t *handle, uint32_t *count)`

Gets the number of bytes written to the LPUART TX register.

This function gets the number of bytes that have been written to LPUART TX register by DMA.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.
- `count` – Send bytes count.

Return values

- `kStatus_NoTransferInProgress` – No send in progress.
- `kStatus_InvalidArgument` – Parameter is invalid.
- `kStatus_Success` – Get successfully through the parameter count;

status_t LPUART_TransferGetReceiveCountDMA(LPUART_Type *base, *lpuart_dma_handle_t* *handle, uint32_t *count)

Gets the number of received bytes.

This function gets the number of received bytes.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- count – Receive bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

void LPUART_TransferDMAHandleIRQ(LPUART_Type *base, void *lpuartDmaHandle)

LPUART DMA IRQ handle function.

This function handles the LPUART tx complete IRQ request and invoke user callback.

Note: This function is used as default IRQ handler by double weak mechanism. If user's specific IRQ handler is implemented, make sure this function is invoked in the handler.

Parameters

- base – LPUART peripheral base address.
- lpuartDmaHandle – LPUART handle pointer.

FSL_LPUART_DMA_DRIVER_VERSION

LPUART DMA driver version.

typedef struct *lpuart_dma_handle* lpuart_dma_handle_t

typedef void (*lpuart_dma_transfer_callback_t)(LPUART_Type *base, *lpuart_dma_handle_t* *handle, *status_t* status, void *userData)

LPUART transfer callback function.

struct *lpuart_dma_handle*

#include <fsl_lpuart_dma.h> LPUART DMA handle.

Public Members

lpuart_dma_transfer_callback_t callback

Callback function.

void *userData

LPUART callback function parameter.

size_t rxDataSizeAll

Size of the data to receive.

size_t txDataSizeAll

Size of the data to send out.

dma_handle_t *txDmaHandle

The DMA TX channel used.

dma_handle_t *rxDmaHandle

The DMA RX channel used.

volatile uint8_t txState

TX transfer state.

volatile uint8_t rxState

RX transfer state

2.30 LPUART Driver

static inline void LPUART_SoftwareReset(LPUART_Type *base)

Resets the LPUART using software.

This function resets all internal logic and registers except the Global Register. Remains set until cleared by software.

Parameters

- base – LPUART peripheral base address.

status_t LPUART_Init(LPUART_Type *base, const *lpuart_config_t* *config, uint32_t srcClock_Hz)

Initializes an LPUART instance with the user configuration structure and the peripheral clock.

This function configures the LPUART module with user-defined settings. Call the LPUART_GetDefaultConfig() function to configure the configuration structure and get the default configuration. The example below shows how to use this API to configure the LPUART.

```
lpuart_config_t lpuartConfig;
lpuartConfig.baudRate_Bps = 115200U;
lpuartConfig.parityMode = kLPUART_ParityDisabled;
lpuartConfig.dataBitsCount = kLPUART_EightDataBits;
lpuartConfig.isMsb = false;
lpuartConfig.stopBitCount = kLPUART_OneStopBit;
lpuartConfig.txFifoWatermark = 0;
lpuartConfig.rxFifoWatermark = 1;
LPUART_Init(LPUART1, &lpuartConfig, 20000000U);
```

Parameters

- base – LPUART peripheral base address.
- config – Pointer to a user-defined configuration structure.
- srcClock_Hz – LPUART clock source frequency in HZ.

Return values

- kStatus_LPUART_BaudrateNotSupport – Baudrate is not support in current clock source.
- kStatus_Success – LPUART initialize succeed

status_t LPUART_Deinit(LPUART_Type *base)

Deinitializes a LPUART instance.

This function waits for transmit to complete, disables TX and RX, and disables the LPUART clock.

Parameters

- base – LPUART peripheral base address.

Return values

- kStatus__Success – Deinit is success.
- kStatus__LPUART_Timeout – Timeout during deinit.

void LPUART_GetDefaultConfig(*lpuart_config_t* *config)

Gets the default configuration structure.

This function initializes the LPUART configuration structure to a default value. The default values are: lpuartConfig->baudRate_Bps = 115200U; lpuartConfig->parityMode = kLPUART_ParityDisabled; lpuartConfig->dataBitsCount = kLPUART_EightDataBits; lpuartConfig->isMsb = false; lpuartConfig->stopBitCount = kLPUART_OneStopBit; lpuartConfig->txFifoWatermark = 0; lpuartConfig->rxFifoWatermark = 1; lpuartConfig->rxIdleType = kLPUART_IdleTypeStartBit; lpuartConfig->rxIdleConfig = kLPUART_IdleCharacter1; lpuartConfig->enableTx = false; lpuartConfig->enableRx = false;

Parameters

- config – Pointer to a configuration structure.

status_t LPUART_SetBaudRate(LPUART_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)

Sets the LPUART instance baudrate.

This function configures the LPUART module baudrate. This function is used to update the LPUART module baudrate after the LPUART module is initialized by the LPUART_Init.

```
LPUART_SetBaudRate(LPUART1, 115200U, 20000000U);
```

Parameters

- base – LPUART peripheral base address.
- baudRate_Bps – LPUART baudrate to be set.
- srcClock_Hz – LPUART clock source frequency in HZ.

Return values

- kStatus__LPUART_BaudrateNotSupport – Baudrate is not supported in the current clock source.
- kStatus__Success – Set baudrate succeeded.

void LPUART_Enable9bitMode(LPUART_Type *base, bool enable)

Enable 9-bit data mode for LPUART.

This function set the 9-bit mode for LPUART module. The 9th bit is not used for parity thus can be modified by user.

Parameters

- base – LPUART peripheral base address.
- enable – true to enable, false to disable.

static inline void LPUART_SetMatchAddress(LPUART_Type *base, uint16_t address1, uint16_t address2)

Set the LPUART address.

This function configures the address for LPUART module that works as slave in 9-bit data mode. One or two address fields can be configured. When the address field's match enable

bit is set, the frame it receives with MSB being 1 is considered as an address frame, otherwise it is considered as data frame. Once the address frame matches one of slave's own addresses, this slave is addressed. This address frame and its following data frames are stored in the receive buffer; otherwise the frames will be discarded. To un-address a slave, just send an address frame with unmatched address.

Note: Any LPUART instance joined in the multi-slave system can work as slave. The position of the address mark is the same as the parity bit when parity is enabled for 8 bit and 9 bit data formats.

Parameters

- base – LPUART peripheral base address.
- address1 – LPUART slave address1.
- address2 – LPUART slave address2.

```
static inline void LPUART__EnableMatchAddress(LPUART_Type *base, bool match1, bool match2)
```

Enable the LPUART match address feature.

Parameters

- base – LPUART peripheral base address.
- match1 – true to enable match address1, false to disable.
- match2 – true to enable match address2, false to disable.

```
static inline void LPUART__SetRxFifoWatermark(LPUART_Type *base, uint8_t water)
```

Sets the rx FIFO watermark.

Parameters

- base – LPUART peripheral base address.
- water – Rx FIFO watermark.

```
static inline void LPUART__SetTxFifoWatermark(LPUART_Type *base, uint8_t water)
```

Sets the tx FIFO watermark.

Parameters

- base – LPUART peripheral base address.
- water – Tx FIFO watermark.

```
static inline void LPUART__TransferEnable16Bit(lpuart_handle_t *handle, bool enable)
```

Sets the LPUART using 16bit transmit, only for 9bit or 10bit mode.

This function Enable 16bit Data transmit in *lpuart_handle_t*.

Parameters

- handle – LPUART handle pointer.
- enable – true to enable, false to disable.

```
uint32_t LPUART__GetStatusFlags(LPUART_Type *base)
```

Gets LPUART status flags.

This function gets all LPUART status flags. The flags are returned as the logical OR value of the enumerators *_lpuart_flags*. To check for a specific status, compare the return value with enumerators in the *_lpuart_flags*. For example, to check whether the TX is empty:

```
if (kLPUART_TxDataRegEmptyFlag & LPUART_GetStatusFlags(LPUART1))
{
    ...
}
```

Parameters

- base – LPUART peripheral base address.

Returns

LPUART status flags which are ORed by the enumerators in the `_lpuart_flags`.

`status_t` LPUART_ClearStatusFlags(LPUART_Type *base, uint32_t mask)

Clears status flags with a provided mask.

This function clears LPUART status flags with a provided mask. Automatically cleared flags can't be cleared by this function. Flags that can only be cleared or set by hardware are: `kLPUART_TxDataRegEmptyFlag`, `kLPUART_TransmissionCompleteFlag`, `kLPUART_RxDataRegFullFlag`, `kLPUART_RxActiveFlag`, `kLPUART_NoiseErrorFlag`, `kLPUART_ParityErrorFlag`, `kLPUART_TxFifoEmptyFlag`, `kLPUART_RxFifoEmptyFlag`. Note: This API should be called when the Tx/Rx is idle, otherwise it takes no effects.

Parameters

- base – LPUART peripheral base address.
- mask – the status flags to be cleared. The user can use the enumerators in the `_lpuart_status_flag_t` to do the OR operation and get the mask.

Return values

- `kStatus_LPUART_FlagCannotClearManually` – The flag can't be cleared by this function but it is cleared automatically by hardware.
- `kStatus_Success` – Status in the mask are cleared.

Returns

0 succeed, others failed.

`void` LPUART_EnableInterrupts(LPUART_Type *base, uint32_t mask)

Enables LPUART interrupts according to a provided mask.

This function enables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See the `_lpuart_interrupt_enable`. This example shows how to enable TX empty interrupt and RX full interrupt:

```
LPUART_EnableInterrupts(LPUART1, kLPUART_TxDataRegEmptyInterruptEnable | kLPUART_
↳ RxDataRegFullInterruptEnable);
```

Parameters

- base – LPUART peripheral base address.
- mask – The interrupts to enable. Logical OR of `_lpuart_interrupt_enable`.

`void` LPUART_DisableInterrupts(LPUART_Type *base, uint32_t mask)

Disables LPUART interrupts according to a provided mask.

This function disables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See `_lpuart_interrupt_enable`. This example shows how to disable the TX empty interrupt and RX full interrupt:

```
LPUART_DisableInterrupts(LPUART1, kLPUART_TxDataRegEmptyInterruptEnable | kLPUART_
↳ RxDataRegFullInterruptEnable);
```


Parameters

- base – LPUART peripheral base address.
- mask – The interrupts to disable. Logical OR of `_lpuart_interrupt_enable`.

`uint32_t LPUART_GetEnabledInterrupts(LPUART_Type *base)`

Gets enabled LPUART interrupts.

This function gets the enabled LPUART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators `_lpuart_interrupt_enable`. To check a specific interrupt enable status, compare the return value with enumerators in `_lpuart_interrupt_enable`. For example, to check whether the TX empty interrupt is enabled:

```
uint32_t enabledInterrupts = LPUART_GetEnabledInterrupts(LPUART1);

if (kLPUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)
{
    ...
}
```

Parameters

- base – LPUART peripheral base address.

Returns

LPUART interrupt flags which are logical OR of the enumerators in `_lpuart_interrupt_enable`.

`static inline uintptr_t LPUART_GetDataRegisterAddress(LPUART_Type *base)`

Gets the LPUART data register address.

This function returns the LPUART data register address, which is mainly used by the DMA/eDMA.

Parameters

- base – LPUART peripheral base address.

Returns

LPUART data register addresses which are used both by the transmitter and receiver.

`static inline void LPUART_EnableTxDMA(LPUART_Type *base, bool enable)`

Enables or disables the LPUART transmitter DMA request.

This function enables or disables the transmit data register empty flag, `STAT[TDRE]`, to generate DMA requests.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

`static inline void LPUART_EnableRxDMA(LPUART_Type *base, bool enable)`

Enables or disables the LPUART receiver DMA.

This function enables or disables the receiver data register full flag, `STAT[RDRF]`, to generate DMA requests.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

```
uint32_t LPUART_GetInstance(LPUART_Type *base)
```

Get the LPUART instance from peripheral base address.

Parameters

- base – LPUART peripheral base address.

Returns

LPUART instance.

```
static inline void LPUART_EnableTx(LPUART_Type *base, bool enable)
```

Enables or disables the LPUART transmitter.

This function enables or disables the LPUART transmitter.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

```
static inline void LPUART_EnableRx(LPUART_Type *base, bool enable)
```

Enables or disables the LPUART receiver.

This function enables or disables the LPUART receiver.

Parameters

- base – LPUART peripheral base address.
- enable – True to enable, false to disable.

```
static inline void LPUART_WriteByte(LPUART_Type *base, uint8_t data)
```

Writes to the transmitter register.

This function writes data to the transmitter register directly. The upper layer must ensure that the TX register is empty or that the TX FIFO has room before calling this function.

Parameters

- base – LPUART peripheral base address.
- data – Data write to the TX register.

```
static inline uint8_t LPUART_ReadByte(LPUART_Type *base)
```

Reads the receiver register.

This function reads data from the receiver register directly. The upper layer must ensure that the receiver register is full or that the RX FIFO has data before calling this function.

Parameters

- base – LPUART peripheral base address.

Returns

Data read from data register.

```
static inline uint8_t LPUART_GetRxFifoCount(LPUART_Type *base)
```

Gets the rx FIFO data count.

Parameters

- base – LPUART peripheral base address.

Returns

rx FIFO data count.

static inline uint8_t LPUART_GetTxFifoCount(LPUART_Type *base)

Gets the tx FIFO data count.

Parameters

- base – LPUART peripheral base address.

Returns

tx FIFO data count.

void LPUART_SendAddress(LPUART_Type *base, uint8_t address)

Transmit an address frame in 9-bit data mode.

Parameters

- base – LPUART peripheral base address.
- address – LPUART slave address.

status_t LPUART_WriteBlocking(LPUART_Type *base, const uint8_t *data, size_t length)

Writes to the transmitter register using a blocking method.

This function polls the transmitter register, first waits for the register to be empty or TX FIFO to have room, and writes data to the transmitter buffer, then waits for the data to be sent out to the bus.

Parameters

- base – LPUART peripheral base address.
- data – Start address of the data to write.
- length – Size of the data to write.

Return values

- kStatus_LPUART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully wrote all data.

status_t LPUART_WriteBlocking16bit(LPUART_Type *base, const uint16_t *data, size_t length)

Writes to the transmitter register using a blocking method in 9bit or 10bit mode.

Note: This function only support 9bit or 10bit transfer. Please make sure only 10bit of data is valid and other bits are 0.

Parameters

- base – LPUART peripheral base address.
- data – Start address of the data to write.
- length – Size of the data to write.

Return values

- kStatus_LPUART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully wrote all data.

status_t LPUART_ReadBlocking(LPUART_Type *base, uint8_t *data, size_t length)

Reads the receiver data register using a blocking method.

This function polls the receiver register, waits for the receiver register full or receiver FIFO has data, and reads data from the RX register.

Parameters

- base – LPUART peripheral base address.

- data – Start address of the buffer to store the received data.
- length – Size of the buffer.

Return values

- kStatus_LPUART_RxHardwareOverrun – Receiver overrun happened while receiving data.
- kStatus_LPUART_NoiseError – Noise error happened while receiving data.
- kStatus_LPUART_FramingError – Framing error happened while receiving data.
- kStatus_LPUART_ParityError – Parity error happened while receiving data.
- kStatus_LPUART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully received all data.

status_t LPUART_ReadBlocking16bit(LPUART_Type *base, uint16_t *data, size_t length)

Reads the receiver data register in 9bit or 10bit mode.

Note: This function only support 9bit or 10bit transfer.

Parameters

- base – LPUART peripheral base address.
- data – Start address of the buffer to store the received data by 16bit, only 10bit is valid.
- length – Size of the buffer.

Return values

- kStatus_LPUART_RxHardwareOverrun – Receiver overrun happened while receiving data.
- kStatus_LPUART_NoiseError – Noise error happened while receiving data.
- kStatus_LPUART_FramingError – Framing error happened while receiving data.
- kStatus_LPUART_ParityError – Parity error happened while receiving data.
- kStatus_LPUART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully received all data.

void LPUART_TransferCreateHandle(LPUART_Type *base, *lpuart_handle_t* *handle, *lpuart_transfer_callback_t* callback, void *userData)

Initializes the LPUART handle.

This function initializes the LPUART handle, which can be used for other LPUART transactional APIs. Usually, for a specified LPUART instance, call this API once to get the initialized handle.

The LPUART driver supports the “background” receiving, which means that user can set up an RX ring buffer optionally. Data received is stored into the ring buffer even when the user doesn’t call the LPUART_TransferReceiveNonBlocking() API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly. The ring buffer is disabled if passing NULL as ringBuffer.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- callback – Callback function.
- userData – User data.

status_t LPUART__TransferSendNonBlocking(LPUART_Type *base, *lpuart_handle_t* *handle, *lpuart_transfer_t* *xfer)

Transmits a buffer of data using the interrupt method.

This function send data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data written to the transmitter register. When all data is written to the TX register in the ISR, the LPUART driver calls the callback function and passes the `kStatus_LPUART_TxIdle` as status parameter.

Note: The `kStatus_LPUART_TxIdle` is passed to the upper layer when all data are written to the TX register. However, there is no check to ensure that all the data sent out. Before disabling the TX, check the `kLPUART_TransmissionCompleteFlag` to ensure that the transmit is finished.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- xfer – LPUART transfer structure, see `lpuart_transfer_t`.

Return values

- `kStatus__Success` – Successfully start the data transmission.
- `kStatus_LPUART_TxBusy` – Previous transmission still not finished, data not all written to the TX register.
- `kStatus_InvalidArgument` – Invalid argument.

void LPUART__TransferStartRingBuffer(LPUART_Type *base, *lpuart_handle_t* *handle, *uint8_t* *ringBuffer, *size_t* ringBufferSize)

Sets up the RX ring buffer.

This function sets up the RX ring buffer to a specific UART handle.

When the RX ring buffer is used, data received is stored into the ring buffer even when the user doesn't call the `UART_TransferReceiveNonBlocking()` API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly.

Note: When using RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, then only 31 bytes are used for saving data.

Parameters

- base – LPUART peripheral base address.
- handle – LPUART handle pointer.
- ringBuffer – Start address of ring buffer for background receiving. Pass NULL to disable the ring buffer.
- ringBufferSize – size of the ring buffer.

`void LPUART__TransferStopRingBuffer(LPUART_Type *base, lpuart_handle_t *handle)`

Aborts the background transfer and uninstalls the ring buffer.

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.

`size_t LPUART__TransferGetRxRingBufferLength(LPUART_Type *base, lpuart_handle_t *handle)`

Get the length of received data in RX ring buffer.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.

Returns

Length of received data in RX ring buffer.

`void LPUART__TransferAbortSend(LPUART_Type *base, lpuart_handle_t *handle)`

Aborts the interrupt-driven data transmit.

This function aborts the interrupt driven data sending. The user can get the `remainBtyes` to find out how many bytes are not sent out.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.

`status_t LPUART__TransferGetSendCount(LPUART_Type *base, lpuart_handle_t *handle, uint32_t *count)`

Gets the number of bytes that have been sent out to bus.

This function gets the number of bytes that have been sent out to bus by an interrupt method.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.
- `count` – Send bytes count.

Return values

- `kStatus_NoTransferInProgress` – No send in progress.
- `kStatus_InvalidArgument` – Parameter is invalid.
- `kStatus_Success` – Get successfully through the parameter count;

`status_t LPUART__TransferReceiveNonBlocking(LPUART_Type *base, lpuart_handle_t *handle, lpuart_transfer_t *xfer, size_t *receivedBytes)`

Receives a buffer of data using the interrupt method.

This function receives data using an interrupt method. This is a non-blocking function which returns without waiting to ensure that all data are received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter `receivedBytes` shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is not enough for read, the receive request is saved by the LPUART driver. When the new data arrives, the receive request is serviced first. When all data is received, the LPUART driver notifies the upper layer through a callback function and passes a status parameter `kStatus_UART_RxIdle`. For example, the upper layer needs 10 bytes but there are only 5

bytes in ring buffer. The 5 bytes are copied to `xfer->data`, which returns with the parameter `receivedBytes` set to 5. For the remaining 5 bytes, the newly arrived data is saved from `xfer->data[5]`. When 5 bytes are received, the LPUART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to `xfer->data`. When all data is received, the upper layer is notified.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.
- `xfer` – LPUART transfer structure, see `uart_transfer_t`.
- `receivedBytes` – Bytes received from the ring buffer directly.

Return values

- `kStatus__Success` – Successfully queue the transfer into the transmit queue.
- `kStatus__LPUART_RxBusy` – Previous receive request is not finished.
- `kStatus__InvalidArgument` – Invalid argument.

`void LPUART_TransferAbortReceive(LPUART_Type *base, lpuart_handle_t *handle)`

Aborts the interrupt-driven data receiving.

This function aborts the interrupt-driven data receiving. The user can get the `remainBytes` to find out how many bytes not received yet.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.

`status_t LPUART_TransferGetReceiveCount(LPUART_Type *base, lpuart_handle_t *handle, uint32_t *count)`

Gets the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- `base` – LPUART peripheral base address.
- `handle` – LPUART handle pointer.
- `count` – Receive bytes count.

Return values

- `kStatus__NoTransferInProgress` – No receive in progress.
- `kStatus__InvalidArgument` – Parameter is invalid.
- `kStatus__Success` – Get successfully through the parameter `count`;

`void LPUART_TransferHandleIRQ(LPUART_Type *base, void *irqHandle)`

LPUART IRQ handle function.

This function handles the LPUART transmit and receive IRQ request.

Parameters

- `base` – LPUART peripheral base address.
- `irqHandle` – LPUART handle pointer.

void LPUART_TransferHandleErrorIRQ(LPUART_Type *base, void *irqHandle)

LPUART Error IRQ handle function.

This function handles the LPUART error IRQ request.

Parameters

- base – LPUART peripheral base address.
- irqHandle – LPUART handle pointer.

void LPUART_DriverIRQHandler(uint32_t instance)

LPUART driver IRQ handler common entry.

This function provides the common IRQ request entry for LPUART.

Parameters

- instance – LPUART instance.

FSL_LPUART_DRIVER_VERSION

LPUART driver version.

Error codes for the LPUART driver.

Values:

enumerator kStatus_LPUART_TxBusy
TX busy

enumerator kStatus_LPUART_RxBusy
RX busy

enumerator kStatus_LPUART_TxIdle
LPUART transmitter is idle.

enumerator kStatus_LPUART_RxIdle
LPUART receiver is idle.

enumerator kStatus_LPUART_TxWatermarkTooLarge
TX FIFO watermark too large

enumerator kStatus_LPUART_RxWatermarkTooLarge
RX FIFO watermark too large

enumerator kStatus_LPUART_FlagCannotClearManually
Some flag can't manually clear

enumerator kStatus_LPUART_Error
Error happens on LPUART.

enumerator kStatus_LPUART_RxRingBufferOverrun
LPUART RX software ring buffer overrun.

enumerator kStatus_LPUART_RxHardwareOverrun
LPUART RX receiver overrun.

enumerator kStatus_LPUART_NoiseError
LPUART noise error.

enumerator kStatus_LPUART_FramingError
LPUART framing error.

enumerator kStatus_LPUART_ParityError
LPUART parity error.

enumerator kStatus_LPUART_BaudrateNotSupport
Baudrate is not support in current clock source

enumerator kStatus_LPUART_IdleLineDetected
IDLE flag.

enumerator kStatus_LPUART_Timeout
LPUART times out.

enum _lpuart_parity_mode
LPUART parity mode.

Values:

enumerator kLPUART_ParityDisabled
Parity disabled

enumerator kLPUART_ParityEven
Parity enabled, type even, bit setting: PE|PT = 10

enumerator kLPUART_ParityOdd
Parity enabled, type odd, bit setting: PE|PT = 11

enum _lpuart_data_bits
LPUART data bits count.

Values:

enumerator kLPUART_EightDataBits
Eight data bit

enumerator kLPUART_SevenDataBits
Seven data bit

enum _lpuart_stop_bit_count
LPUART stop bit count.

Values:

enumerator kLPUART_OneStopBit
One stop bit

enumerator kLPUART_TwoStopBit
Two stop bits

enum _lpuart_transmit_cts_source
LPUART transmit CTS source.

Values:

enumerator kLPUART_CtsSourcePin
CTS resource is the LPUART_CTS pin.

enumerator kLPUART_CtsSourceMatchResult
CTS resource is the match result.

enum _lpuart_transmit_cts_config
LPUART transmit CTS configure.

Values:

enumerator kLPUART_CtsSampleAtStart
CTS input is sampled at the start of each character.

enumerator kLPUART_CtsSampleAtIdle

CTS input is sampled when the transmitter is idle

enum __lpuart_idle_type_select

LPUART idle flag type defines when the receiver starts counting.

Values:

enumerator kLPUART_IdleTypeStartBit

Start counting after a valid start bit.

enumerator kLPUART_IdleTypeStopBit

Start counting after a stop bit.

enum __lpuart_idle_config

LPUART idle detected configuration. This structure defines the number of idle characters that must be received before the IDLE flag is set.

Values:

enumerator kLPUART_IdleCharacter1

the number of idle characters.

enumerator kLPUART_IdleCharacter2

the number of idle characters.

enumerator kLPUART_IdleCharacter4

the number of idle characters.

enumerator kLPUART_IdleCharacter8

the number of idle characters.

enumerator kLPUART_IdleCharacter16

the number of idle characters.

enumerator kLPUART_IdleCharacter32

the number of idle characters.

enumerator kLPUART_IdleCharacter64

the number of idle characters.

enumerator kLPUART_IdleCharacter128

the number of idle characters.

enum __lpuart_interrupt_enable

LPUART interrupt configuration structure, default settings all disabled.

This structure contains the settings for all LPUART interrupt configurations.

Values:

enumerator kLPUART_LinBreakInterruptEnable

LIN break detect. bit 7

enumerator kLPUART_RxActiveEdgeInterruptEnable

Receive Active Edge. bit 6

enumerator kLPUART_TxDataRegEmptyInterruptEnable

Transmit data register empty. bit 23

enumerator kLPUART_TransmissionCompleteInterruptEnable

Transmission complete. bit 22

enumerator kLPUART_RxDataRegFullInterruptEnable
Receiver data register full. bit 21

enumerator kLPUART_IdleLineInterruptEnable
Idle line. bit 20

enumerator kLPUART_RxOverrunInterruptEnable
Receiver Overrun. bit 27

enumerator kLPUART_NoiseErrorInterruptEnable
Noise error flag. bit 26

enumerator kLPUART_FramingErrorInterruptEnable
Framing error flag. bit 25

enumerator kLPUART_ParityErrorInterruptEnable
Parity error flag. bit 24

enumerator kLPUART_Match1InterruptEnable
Parity error flag. bit 15

enumerator kLPUART_Match2InterruptEnable
Parity error flag. bit 14

enumerator kLPUART_TxFifoOverflowInterruptEnable
Transmit FIFO Overflow. bit 9

enumerator kLPUART_RxFifoUnderflowInterruptEnable
Receive FIFO Underflow. bit 8

enumerator kLPUART_AllInterruptEnable

enum _lpuart_flags

LPUART status flags.

This provides constants for the LPUART status flags for use in the LPUART functions.

Values:

enumerator kLPUART_TxDataRegEmptyFlag
Transmit data register empty flag, sets when transmit buffer is empty. bit 23

enumerator kLPUART_TransmissionCompleteFlag
Transmission complete flag, sets when transmission activity complete. bit 22

enumerator kLPUART_RxDataRegFullFlag
Receive data register full flag, sets when the receive data buffer is full. bit 21

enumerator kLPUART_IdleLineFlag
Idle line detect flag, sets when idle line detected. bit 20

enumerator kLPUART_RxOverrunFlag
Receive Overrun, sets when new data is received before data is read from receive register. bit 19

enumerator kLPUART_NoiseErrorFlag
Receive takes 3 samples of each received bit. If any of these samples differ, noise flag sets. bit 18

enumerator kLPUART_FramingErrorFlag
Frame error flag, sets if logic 0 was detected where stop bit expected. bit 17

enumerator kLPUART_ParityErrorFlag
If parity enabled, sets upon parity error detection. bit 16

enumerator `kLPUART_LinBreakFlag`

LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled. bit 31

enumerator `kLPUART_RxActiveEdgeFlag`

Receive pin active edge interrupt flag, sets when active edge detected. bit 30

enumerator `kLPUART_RxActiveFlag`

Receiver Active Flag (RAF), sets at beginning of valid start. bit 24

enumerator `kLPUART_DataMatch1Flag`

The next character to be read from LPUART_DATA matches MA1. bit 15

enumerator `kLPUART_DataMatch2Flag`

The next character to be read from LPUART_DATA matches MA2. bit 14

enumerator `kLPUART_TxFifoEmptyFlag`

TXEMPT bit, sets if transmit buffer is empty. bit 7

enumerator `kLPUART_RxFifoEmptyFlag`

RXEMPT bit, sets if receive buffer is empty. bit 6

enumerator `kLPUART_TxFifoOverflowFlag`

TXOF bit, sets if transmit buffer overflow occurred. bit 1

enumerator `kLPUART_RxFifoUnderflowFlag`

RXUF bit, sets if receive buffer underflow occurred. bit 0

enumerator `kLPUART_AllClearFlags`

enumerator `kLPUART_AllFlags`

typedef enum *lpuart_parity_mode* lpuart_parity_mode_t

LPUART parity mode.

typedef enum *lpuart_data_bits* lpuart_data_bits_t

LPUART data bits count.

typedef enum *lpuart_stop_bit_count* lpuart_stop_bit_count_t

LPUART stop bit count.

typedef enum *lpuart_transmit_cts_source* lpuart_transmit_cts_source_t

LPUART transmit CTS source.

typedef enum *lpuart_transmit_cts_config* lpuart_transmit_cts_config_t

LPUART transmit CTS configure.

typedef enum *lpuart_idle_type_select* lpuart_idle_type_select_t

LPUART idle flag type defines when the receiver starts counting.

typedef enum *lpuart_idle_config* lpuart_idle_config_t

LPUART idle detected configuration. This structure defines the number of idle characters that must be received before the IDLE flag is set.

typedef struct *lpuart_config* lpuart_config_t

LPUART configuration structure.

typedef struct *lpuart_transfer* lpuart_transfer_t

LPUART transfer structure.

typedef struct *lpuart_handle* lpuart_handle_t

```
typedef void (*lpuart_transfer_callback_t)(LPUART_Type *base, lpuart_handle_t *handle,  
status_t status, void *userData)
```

LPUART transfer callback function.

```
typedef void (*lpuart_isr_t)(LPUART_Type *base, void *handle)
```

```
void *_s_lpuartHandle[]
```

```
const IRQn_Type s_lpuartTxIRQ[]
```

```
lpuart_isr_t s_lpuartIsr[]
```

```
UART_RETRY_TIMES
```

Retry times for waiting flag.

```
struct _lpuart_config
```

#include <fsl_lpuart.h> LPUART configuration structure.

Public Members

```
uint32_t baudRate_Bps
```

LPUART baud rate

```
lpuart_parity_mode_t parityMode
```

Parity mode, disabled (default), even, odd

```
lpuart_data_bits_t dataBitsCount
```

Data bits count, eight (default), seven

```
bool isMsb
```

Data bits order, LSB (default), MSB

```
lpuart_stop_bit_count_t stopBitCount
```

Number of stop bits, 1 stop bit (default) or 2 stop bits

```
uint8_t txFifoWatermark
```

TX FIFO watermark

```
uint8_t rxFifoWatermark
```

RX FIFO watermark

```
bool enableRxRTS
```

RX RTS enable

```
bool enableTxCTS
```

TX CTS enable

```
lpuart_transmit_cts_source_t txCtsSource
```

TX CTS source

```
lpuart_transmit_cts_config_t txCtsConfig
```

TX CTS configure

```
lpuart_idle_type_select_t rxIdleType
```

RX IDLE type.

```
lpuart_idle_config_t rxIdleConfig
```

RX IDLE configuration.

```
bool enableTx
```

Enable TX

bool enableRx
 Enable RX

struct __lpuart__transfer
 #include <fsl_lpuart.h> LPUART transfer structure.

Public Members

size_t dataSize
 The byte count to be transfer.

struct __lpuart__handle
 #include <fsl_lpuart.h> LPUART handle structure.

Public Members

volatile size_t txDataSize
 Size of the remaining data to send.

size_t txDataSizeAll
 Size of the data to send out.

volatile size_t rxDataSize
 Size of the remaining data to receive.

size_t rxDataSizeAll
 Size of the data to receive.

size_t rxRingBufferSize
 Size of the ring buffer.

volatile uint16_t rxRingBufferHead
 Index for the driver to store received data into ring buffer.

volatile uint16_t rxRingBufferTail
 Index for the user to get data from the ring buffer.

lpuart_transfer_callback_t callback
 Callback function.

void *userData
 LPUART callback function parameter.

volatile uint8_t txState
 TX transfer state.

volatile uint8_t rxState
 RX transfer state.

bool isSevenDataBits
 Seven data bits flag.

bool is16bitData
 16bit data bits flag, only used for 9bit or 10bit data

union __unnamed22__

Public Members

uint8_t *data

The buffer of data to be transfer.

uint8_t *rxData

The buffer to receive data.

uint16_t *rxData16

The buffer to receive data.

const uint8_t *txData

The buffer of data to be sent.

const uint16_t *txData16

The buffer of data to be sent.

union __unnamed24__

Public Members

const uint8_t *volatile txData

Address of remaining data to send.

const uint16_t *volatile txData16

Address of remaining data to send.

union __unnamed26__

Public Members

uint8_t *volatile rxData

Address of remaining data to receive.

uint16_t *volatile rxData16

Address of remaining data to receive.

union __unnamed28__

Public Members

uint8_t *rxRingBuffer

Start address of the receiver ring buffer.

uint16_t *rxRingBuffer16

Start address of the receiver ring buffer.

2.31 MCM: Miscellaneous Control Module

FSL_MCM_DRIVER_VERSION

MCM driver version.

Enum _mcm_interrupt_flag. Interrupt status flag mask. .

Values:

enumerator kMCM_CacheWriteBuffer
Cache Write Buffer Error Enable.

enumerator kMCM_ParityError
Cache Parity Error Enable.

enumerator kMCM_FPUInvalidOperation
FPU Invalid Operation Interrupt Enable.

enumerator kMCM_FPUDivideByZero
FPU Divide-by-zero Interrupt Enable.

enumerator kMCM_FPUOverflow
FPU Overflow Interrupt Enable.

enumerator kMCM_FPUUnderflow
FPU Underflow Interrupt Enable.

enumerator kMCM_FPUInexact
FPU Inexact Interrupt Enable.

enumerator kMCM_FPUInputDenormalInterrupt
FPU Input Denormal Interrupt Enable.

typedef union *_mcm_buffer_fault_attribute* mcm_buffer_fault_attribute_t
The union of buffer fault attribute.

typedef union *_mcm_lmem_fault_attribute* mcm_lmem_fault_attribute_t
The union of LMEM fault attribute.

static inline void MCM_EnableCrossbarRoundRobin(MCM_Type *base, bool enable)
Enables/Disables crossbar round robin.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable crossbar round robin.
 - **true** Enable crossbar round robin.
 - **false** disable crossbar round robin.

static inline void MCM_EnableInterruptStatus(MCM_Type *base, uint32_t mask)
Enables the interrupt.

Parameters

- base – MCM peripheral base address.
- mask – Interrupt status flags mask(*_mcm_interrupt_flag*).

static inline void MCM_DisableInterruptStatus(MCM_Type *base, uint32_t mask)
Disables the interrupt.

Parameters

- base – MCM peripheral base address.
- mask – Interrupt status flags mask(*_mcm_interrupt_flag*).

static inline uint16_t MCM_GetInterruptStatus(MCM_Type *base)
Gets the Interrupt status .

Parameters

- base – MCM peripheral base address.

static inline void MCM_ClearCacheWriteBufferErrorStatus(MCM_Type *base)

Clears the Interrupt status .

Parameters

- base – MCM peripheral base address.

static inline uint32_t MCM_GetBufferFaultAddress(MCM_Type *base)

Gets buffer fault address.

Parameters

- base – MCM peripheral base address.

static inline void MCM_GetBufferFaultAttribute(MCM_Type *base, *mcm_buffer_fault_attribute_t* *bufferfault)

Gets buffer fault attributes.

Parameters

- base – MCM peripheral base address.
- bufferfault – Structure to store the result.

static inline uint32_t MCM_GetBufferFaultData(MCM_Type *base)

Gets buffer fault data.

Parameters

- base – MCM peripheral base address.

static inline void MCM_LimitCodeCachePeripheralWriteBuffering(MCM_Type *base, bool enable)

Limit code cache peripheral write buffering.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable limit code cache peripheral write buffering.
 - **true** Enable limit code cache peripheral write buffering.
 - **false** disable limit code cache peripheral write buffering.

static inline void MCM_BypassFixedCodeCacheMap(MCM_Type *base, bool enable)

Bypass fixed code cache map.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable bypass fixed code cache map.
 - **true** Enable bypass fixed code cache map.
 - **false** disable bypass fixed code cache map.

static inline void MCM_EnableCodeBusCache(MCM_Type *base, bool enable)

Enables/Disables code bus cache.

Parameters

- base – MCM peripheral base address.
- enable – Used to disable/enable code bus cache.
 - **true** Enable code bus cache.
 - **false** disable code bus cache.

static inline void MCM_ForceCodeCacheToNoAllocation(MCM_Type *base, bool enable)
Force code cache to no allocation.

Parameters

- base – MCM peripheral base address.
- enable – Used to force code cache to allocation or no allocation.
 - **true** Force code cache to no allocation.
 - **false** Force code cache to allocation.

static inline void MCM_EnableCodeCacheWriteBuffer(MCM_Type *base, bool enable)
Enables/Disables code cache write buffer.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable code cache write buffer.
 - **true** Enable code cache write buffer.
 - **false** Disable code cache write buffer.

static inline void MCM_ClearCodeBusCache(MCM_Type *base)
Clear code bus cache.

Parameters

- base – MCM peripheral base address.

static inline void MCM_EnablePcParityFaultReport(MCM_Type *base, bool enable)
Enables/Disables PC Parity Fault Report.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable PC Parity Fault Report.
 - **true** Enable PC Parity Fault Report.
 - **false** disable PC Parity Fault Report.

static inline void MCM_EnablePcParity(MCM_Type *base, bool enable)
Enables/Disables PC Parity.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable PC Parity.
 - **true** Enable PC Parity.
 - **false** disable PC Parity.

static inline void MCM_LockConfigState(MCM_Type *base)
Lock the configuration state.

Parameters

- base – MCM peripheral base address.

static inline void MCM_EnableCacheParityReporting(MCM_Type *base, bool enable)
Enables/Disables cache parity reporting.

Parameters

- base – MCM peripheral base address.

- enable – Used to enable/disable cache parity reporting.
 - **true** Enable cache parity reporting.
 - **false** disable cache parity reporting.

static inline uint32_t MCM_GetLmemFaultAddress(MCM_Type *base)

Gets LMEM fault address.

Parameters

- base – MCM peripheral base address.

static inline void MCM_GetLmemFaultAttribute(MCM_Type *base, mcm_lmem_fault_attribute_t *lmemFault)

Get LMEM fault attributes.

Parameters

- base – MCM peripheral base address.
- lmemFault – Structure to store the result.

static inline uint64_t MCM_GetLmemFaultData(MCM_Type *base)

Gets LMEM fault data.

Parameters

- base – MCM peripheral base address.

MCM_LMFATR_TYPE_MASK

MCM_LMFATR_MODE_MASK

MCM_LMFATR_BUFF_MASK

MCM_LMFATR_CACH_MASK

MCM_ISCR_STAT_MASK

FSL_COMPONENT_ID

union _mcm_buffer_fault_attribute

#include <fsl_mcm.h> The union of buffer fault attribute.

Public Members

uint32_t attribute

Indicates the faulting attributes, when a properly-enabled cache write buffer error interrupt event is detected.

struct _mcm_buffer_fault_attribute._mcm_buffer_fault_attribut attribute_memory

struct _mcm_buffer_fault_attribut

#include <fsl_mcm.h>

Public Members

uint32_t busErrorDataAccessType

Indicates the type of cache write buffer access.

uint32_t busErrorPrivilegeLevel

Indicates the privilege level of the cache write buffer access.

uint32_t busErrorSize

Indicates the size of the cache write buffer access.

uint32_t busErrorAccess

Indicates the type of system bus access.

uint32_t busErrorMasterID

Indicates the crossbar switch bus master number of the captured cache write buffer bus error.

uint32_t busErrorOverrun

Indicates if another cache write buffer bus error is detected.

union _mcm_lmem_fault_attribute

#include <fsl_mcm.h> The union of LMEM fault attribute.

Public Members

uint32_t attribute

Indicates the attributes of the LMEM fault detected.

struct _mcm_lmem_fault_attribute._mcm_lmem_fault_attribut attribute_memory

struct _mcm_lmem_fault_attribut

#include <fsl_mcm.h>

Public Members

uint32_t parityFaultProtectionSignal

Indicates the features of parity fault protection signal.

uint32_t parityFaultMasterSize

Indicates the parity fault master size.

uint32_t parityFaultWrite

Indicates the parity fault is caused by read or write.

uint32_t backdoorAccess

Indicates the LMEM access fault is initiated by core access or backdoor access.

uint32_t parityFaultSyndrome

Indicates the parity fault syndrome.

uint32_t overrun

Indicates the number of faultss.

2.32 MMAU: Memory Mapped Arithmetic Unit

typedef short int frac16_t

Q0.15 fractional

typedef long frac24_t

Q8.23 fractional

typedef long frac32_t

Q0.31 fractional

```
typedef long long frac48_t
```

Q16.47 fractional

```
typedef long long frac64_t
```

Q0.63 fractional

```
static inline void MMAU__EnableDMA(MMAU_Type *base, bool enable)
```

Enable DMA for MMAU module.

Parameters

- base – MMAU peripheral address.
- enable – Mode of DMA access
 - true Enable DMA access
 - false Disable DMA access

```
static inline void MMAU__EnableSupervisorOnly(MMAU_Type *base, bool enable)
```

Enable supervisor only for MMAU module.

Parameters

- base – MMAU peripheral address.
- enable – Mode of MMAU module can be access
 - true MMAU registers can only be access in Supervisor Mode.
 - false MMAU registers can be access in both User Mode or Supervisor Mode.

```
void MMAU__Reset(MMAU_Type *base)
```

Set control/status register into reset state.

This function sets control/status register to a known state. This state is defined in Reference Manual, which is power on reset value. This function must execute in a Supervisor Mode

Parameters

- base – MMAU peripheral address.

```
static inline void MMAU__EnableInterrupts(MMAU_Type *base, uint32_t mask)
```

Enable MMAU interrupts.

This function enables the interrupts related to the mask. Example:

```
MMAU__EnableInterrupts(MMAU, kMMAU__AccumOverflowInterruptEnable | kMMAU__
↪ DivideByZeroInterruptEnable);
```

Parameters

- base – MMAU peripheral address.
- mask – Mask of the interrupt enable to be written (kMMAU__AccumOverflowInterruptEnable | kMMAU__OverflowInterruptEnable | kMMAU__DivideByZeroInterruptEnable)

```
static inline void MMAU__DisableInterrupts(MMAU_Type *base, uint32_t mask)
```

Disable MMAU interrupts.

This function disables the interrupt related to the mask. Example:

```
MMAU__DisableInterrupts(MMAU, kMMAU__AccumOverflowInterruptEnable | kMMAU__
↪ DivideByZeroInterruptEnable);
```

Parameters

- base – MMAU peripheral address.
- mask – Mask of the interrupt enable to be written (kMMAU_AccumOverflowInterruptEnable | kMMAU_OverflowInterruptEnable | kMMAU_DivideByZeroInterruptEnable).

static inline uint32_t MMAU_GetEnabledInterrupts(MMAU_Type *base)

Gets enabled interrupts.

This function gets all interrupt values. Example:

```
uint32_t flags = 0;
...
flags = MMAU_GetEnabledInterrupts(MMAU);
if (flags & kMMAU_AccumOverflowInterruptEnable)
{
    ...
}
```

Parameters

- base – MMAU peripheral address.

Returns

Combination of enabled interrupt

static inline uint32_t MMAU_GetInterruptFlags(MMAU_Type *base)

Get interrupt flags.

This function gets interrupt flags.

Parameters

- base – MMAU peripheral address.

Returns

the mask of these interrupt flag bits.

void MMAU_ClearInterruptFlags(MMAU_Type *base, uint32_t mask)

Clears interrupt flags.

This function clears the interrupt flags. Example, if you want to clear Overflow and Divide-By-Zero interrupt flags:

```
MMAU_ClearInterruptFlags(MMAU, kMMAU_OverflowInterruptFlag | kMMAU_
↳ DivideByZeroInterruptFlag);
```

Parameters

- base – MMAU peripheral address.
- mask – Mask of the asserted interrupt flags (kMMAU_AccumOverflowInterruptFlag | kMMAU_OverflowInterruptFlag | kMMAU_DivideByZeroInterruptFlag).

static inline uint32_t MMAU_GetInstructionFlags(MMAU_Type *base)

Gets the instruction flags.

This function gets the instruction flag. Instruction flags are updated by the MMAU after computation of each instruction. Example:

```
uint32_t flags;
...
flags = MMAU_GetInstructionFlags(MMAU);
if (flags & kMMAU_OverflowInstructionFlag)
{
    ...
}
```

Parameters

- base – MMAU peripheral address.

Returns

Combination of all instruction flags.

`void MMAU_SetInstructionFlags(MMAU_Type *base, uint32_t mask)`

Sets the instruction flags.

This function sets the instruction flags. Example:

```
MMAU_SetInstructionFlags(MMAU, kMMAU_AccumOverflowInstructionFlag | kMMAU_
↪NegativeInstructionFlag);
MMAU_SetInstructionFlags(MMAU, kMMAU_OverflowInstructionFlag | kMMAU_
↪DivideByZeroInstructionFlag);
```

`(kMMAU_AccumOverflowInstructionFlag | kMMAU_OverflowInstructionFlag | kMMAU_DivideByZeroInstr`

Parameters

- base – MMAU peripheral address.
- mask – Mask of the instruction flags to be written

`void MMAU_ClearInstructionFlags(MMAU_Type *base, uint32_t mask)`

Clears instruction flags.

This function clears the instruction flags. Example, if you want to clear Overflow and DivideByZero instruction flags:

```
MMAU_ClearInstructionFlags(MMAU, kMMAU_OverflowInstructionFlag | kMMAU_
↪DivideByZeroInstructionFlag);
```

`(kMMAU_AccumOverflowInstructionFlag | kMMAU_OverflowInstructionFlag | kMMAU_DivideByZeroInstr`

Parameters

- base – MMAU peripheral address.
- mask – Mask of the asserted instruction flags

`static inline void MMAU_ulda_d (register uint64_t dval)`

Load A10 accumulator register of the MMAU by 64-bit unsigned value.

The `ulda_d` function loads A10 accumulator register of the MMAU by 64-bit unsigned value.

Parameters

- dval – `uint64_t` unsigned load value.

`static inline uint64_t MMAU_d_umul_ll (register uint32_t lval1, register uint32_t lval2)`

Multiply two 32-bit unsigned values returning a 64-bit unsigned product.

The `MMAU_d_umul_ll` function multiplies two 32-bit unsigned values returning a 64-bit unsigned product.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- lval1 – `uint32_t` unsigned value.
- lval2 – `uint32_t` unsigned value.

Returns

uint64_t unsigned value after multiply.

static inline uint64_t MMAU_d_umul_dl (register uint64_t dval, register uint32_t lval)

Multiply 64-bit unsigned value with 32-bit unsigned value returning a 64-bit unsigned product.

The MMAU_d_umul_dl function multiplies 64-bit unsigned value with 32-bit unsigned value returning a 64-bit unsigned product.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – uint64_t unsigned value.
- lval – uint32_t unsigned value.

Returns

uint64_t unsigned value after multiply.

static inline uint64_t MMAU_d_umuls_dl (register uint64_t dval, register uint32_t lval)

Saturating multiply 64-bit unsigned value with 32-bit unsigned value returning saturated 64-bit unsigned product.

The MMAU_d_umuls_dl function multiplies 64-bit unsigned value with 32-bit unsigned value returning saturated 64-bit unsigned product.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – uint64_t unsigned value.
- lval – uint32_t unsigned value.

Returns

uint64_t unsigned value after multiply.

static inline uint64_t MMAU_d_umula_l (register uint32_t lval)

Multiply 32-bit unsigned value with 64-bit unsigned value stored in the A10 register of the MMAU returning a 64-bit unsigned product.

The MMAU_d_umula_l function multiplies 32-bit unsigned value with 64-bit unsigned value stored in the A10 register of the MMAU returning a 64-bit unsigned product.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- lval – uint32_t unsigned value.

Returns

uint64_t unsigned value after multiply.

static inline uint64_t MMAU_d_umulas_l (register uint32_t lval)

Saturating multiply 32-bit unsigned value with 64-bit unsigned value stored in the A10 register of the MMAU returning saturated 64-bit unsigned product.

The MMAU_d_umulas_l function multiplies 32-bit unsigned value with 64-bit unsigned value stored in the A10 register of the MMAU returning saturated 64-bit unsigned product.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- lval – uint32_t unsigned value.

Returns

uint64_t unsigned value after multiply.

static inline uint64_t MMAU_d_umac_ll (register uint32_t lval1, register uint32_t lval2)

Multiply two 32-bit unsigned values and add product with value stored in the A10 register of the MMAU returning a 64-bit unsigned A10 register value.

The MMAU_d_umac_ll function multiplies two 32-bit unsigned values and add product with value stored in the A10 register of the MMAU returning a 64-bit unsigned A10 register value.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- lval1 – uint32_t unsigned value.
- lval2 – uint32_t unsigned value.

Returns

uint64_t unsigned value after multiply.

static inline uint64_t MMAU_d_umacs_ll (register uint32_t lval1, register uint32_t lval2)

Saturating multiply two 32-bit unsigned values and add product with value stored in the A10 register of the MMAU returning a 64-bit unsigned A10 register value.

The MMAU_d_umacs_ll function multiplies two 32-bit unsigned values and add product with value stored in the A10 register of the MMAU returning saturated 64-bit unsigned A10 register value.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- lval1 – uint32_t unsigned value.
- lval2 – uint32_t unsigned value.

Returns

uint64_t unsigned value after multiply.

static inline uint64_t MMAU_d_umac_dl (register uint64_t dval, register uint32_t lval)

Multiply 64-bit unsigned value with 32-bit unsigned value and add product with value stored in the A10 register of the MMAU returning a 64-bit unsigned A10 register value.

The MMAU_d_umac_dl function multiplies 64-bit unsigned value with 32-bit unsigned value and add product with value stored in the A10 register of the MMAU returning a 64-bit unsigned A10 register value.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – uint64_t unsigned value.
- lval – uint32_t unsigned value.

Returns

uint64_t unsigned value after multiply.

static inline uint64_t MMAU_d_umacs_dl (register uint64_t dval, register uint32_t lval)

Saturating multiply 64-bit unsigned value with 32-bit unsigned value and add product with value stored in the A10 register of the MMAU returning saturated 64-bit unsigned A10 register value.

The MMAU_d_umacs_dl function multiplies 64-bit unsigned value with 32-bit unsigned value and add product with value stored in the A10 register of the MMAU returning saturated 64-bit unsigned A10 register value.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – uint64_t unsigned value.
- lval – uint32_t unsigned value.

Returns

uint64_t unsigned value after multiply.

static inline uint64_t MMAU_d_umaca_dl (register uint64_t dval, register uint32_t lval)

Multiply 32-bit unsigned value by value stored in the A10 register of the MMAU and add product with 64-bit unsigned value returning a 64-bit unsigned A10 register value.

The MMAU_d_umaca_dl function multiplies 32-bit unsigned value by value stored in the A10 register of the MMAU and add product with 64-bit unsigned value returning a 64-bit unsigned A10 register value.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – uint64_t unsigned value.
- lval – uint32_t unsigned value.

Returns

uint64_t unsigned value after multiply.

static inline uint64_t MMAU_d_umacas_dl (register uint64_t dval, register uint32_t lval)

Saturating multiply 32-bit unsigned value by value stored in the A10 register of the MMAU and add product with 64-bit unsigned value returning a saturated 64-bit unsigned A10 register value.

The MMAU_d_umacas_dl function multiplies 32-bit unsigned value by value stored in the A10 register of the MMAU and add product with 64-bit unsigned value returning saturated 64-bit unsigned A10 register value.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – uint64_t unsigned value.

- lval – uint32_t unsigned value.

Returns

uint64_t unsigned value after multiply.

static inline uint32_t MMAU_l_udiv_ll (register uint32_t lnum, register uint32_t lden)

Divide two 32-bit unsigned values returning a 32-bit unsigned quotient.

The MMAU_l_udiv_ll function divides two 32-bit unsigned values returning a 32-bit unsigned quotient.

Note: Quotient is stored in A0 register of the MMAU for next computation.

Parameters

- lnum – uint32_t unsigned divisor value.
- lden – uint32_t unsigned dividend value.

Returns

uint32_t unsigned quotient value.

static inline uint64_t MMAU_d_udiv_dl (register uint64_t dnum, register uint32_t lden)

Divide 64-bit unsigned value by 32-bit unsigned value returning a 64-bit unsigned quotient.

The MMAU_d_udiv_dl function divides 64-bit unsigned value by 32-bit unsigned value returning a 64-bit unsigned quotient.

Note: Quotient is stored in A10 register of the MMAU for next computation.

Parameters

- dnum – uint64_t unsigned divisor value.
- lden – uint32_t unsigned dividend value.

Returns

uint64_t unsigned quotient value.

static inline uint64_t MMAU_d_udiv_dd (register uint64_t dnum, register uint64_t dden)

Divide two 64-bit unsigned values returning a 64-bit unsigned quotient.

The MMAU_d_udiv_dd function divides two 64-bit unsigned values returning a 64-bit unsigned quotient.

Note: Quotient is stored in A10 register of the MMAU for next computation.

Parameters

- dnum – uint64_t unsigned divisor value.
- dden – uint64_t unsigned dividend value.

Returns

uint64_t unsigned quotient value.

static inline uint64_t MMAU_d_udiva_l (register uint32_t lden1)

Divide 32-bit unsigned value stored in the A10 register of the MMAU by 32-bit unsigned value returning a 64-bit unsigned quotient.

The MMAU_d_udiva_l function divides 64-bit unsigned value stored in the A10 register of the MMAU by 32-bit unsigned value returning a 64-bit unsigned quotient.

Note: Quotient is stored in A10 register of the MMAU for next computation.

Parameters

- `lden1` – `uint32_t` unsigned dividend value.

Returns

`uint64_t` unsigned quotient value.

`static inline uint64_t MMAU_d_udiva_d (register uint64_t dden1)`

Divide 64-bit unsigned value stored in the A10 register of the MMAU by 64-bit unsigned value returning a 64-bit unsigned quotient.

The `MMAU_d_udiva_d` function divides 64-bit unsigned value stored in the A10 register of the MMAU by 64-bit unsigned value returning a 64-bit unsigned quotient.

Note: Quotient is stored in A10 register of the MMAU for next computation.

Parameters

- `dden1` – `uint64_t` unsigned dividend value.

Returns

`uint64_t` unsigned quotient value.

`static inline uint32_t MMAU_l_usqr_l (register uint32_t lrad)`

Compute and return a 32-bit unsigned square root of the 32-bit unsigned radicand.

The `MMAU_l_usqr_l` function computes and returns a 32-bit unsigned square root of the 32-bit unsigned radicand.

Note: Quotient is stored in A0 register of the MMAU for next computation.

Parameters

- `lrad` – `uint32_t` unsigned radicand.

Returns

`uint32_t` unsigned square root.

`static inline uint32_t MMAU_l_usqr_d (register uint64_t drad)`

Compute and return a 32-bit unsigned square root of the 64-bit unsigned radicand.

The `MMAU_l_usqr_d` function computes and returns a 32-bit unsigned square root of the 64-bit unsigned radicand.

Note: Quotient is stored in A0 register of the MMAU for next computation.

Parameters

- `drad` – `uint64_t` unsigned radicand.

Returns

`uint32_t` unsigned square root.

static inline uint16_t MMAU_s_usqr_l (register uint32_t lrad)

Compute and return a 16-bit unsigned square root of the 32-bit unsigned radicand.

The MMAU_s_usqr_l function computes and returns a 16-bit unsigned square root of the 32-bit unsigned radicand.

Note: Square root is stored in A0 register of the MMAU for next computation.

Parameters

- lrad – uint32 unsigned radicand.

Returns

uint16 unsigned square root.

static inline uint32_t MMAU_l_usqra(void)

Compute and return a 32-bit unsigned square root of the radicand stored in the A10 register of the MMAU.

The MMAU_l_usqra function computes and returns a 32-bit unsigned square root of the radicand stored in the A10 register of the MMAU.

Note: Quotient is stored in A0 register of the MMAU for next computation.

Returns

uint32_t unsigned square root.

static inline void MMAU_sllda_d (register int64_t dval)

Load A10 accumulator register of the MMAU by 64-bit integer value.

The MMAU_sllda_d function loads A10 accumulator register of the MMAU by 64-bit integer value.

Parameters

- dval – int64_t integer value.

static inline int64_t MMAU_d_smul_ll (register int32_t lval1, register int32_t lval2)

Multiply two 32-bit integer values returning a 64-bit integer product.

The MMAU_d_smul_ll function multiplies two 32-bit integer values returning a 64-bit integer product.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- lval1 – int32_t integer value.
- lval2 – int32_t integer value.

Returns

int64_t integer value after multiply.

static inline int64_t MMAU_d_smul_dl (register int64_t dval, register int32_t lval)

Multiply 64-bit integer value with 32-bit integer value returning a 64-bit integer product.

The MMAU_d_smul_dl function multiplies 64-bit integer value with 32-bit integer value returning a 64-bit integer product.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – int64_t integer value.
- lval – int32_t integer value.

Returns

int64_t integer value after multiply.

static inline int64_t MMAU_d_smuls_dl (register int64_t dval, register int32_t lval)

Saturating multiply 64-bit integer value with 32-bit integer value returning saturated 64-bit integer product.

The MMAU_d_smuls_dl function multiplies 64-bit integer value with 32-bit integer value returning saturated 64-bit integer product.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – int64_t integer value.
- lval – int32_t integer value.

Returns

int64_t integer value after multiply.

static inline int64_t MMAU_d_smula_l (register int32_t lval)

Multiply 32-bit integer value with 64-bit integer value stored in the A10 register of the MMAU returning a 64-bit integer product.

The MMAU_d_smula_l function multiplies 32-bit integer value with 64-bit integer value stored in the A10 register of the MMAU returning a 64-bit integer product.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- lval – int32_t integer value.

Returns

int64_t integer value after multiply.

static inline int64_t MMAU_d_smulas_l (register int32_t lval)

Saturating multiply 32-bit integer value with 64-bit integer value stored in the A10 register of the MMAU returning saturated 64-bit integer product.

The MMAU_d_smulas_l function multiplies 32-bit integer value with 64-bit integer value stored in the A10 register of the MMAU returning saturated 64-bit integer product.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- lval – int32_t integer value.

Returns

int64_t integer value after multiply.

```
static inline int64_t MMAU_d_smac_ll (register int32_t lval1, register int32_t lval2)
```

Multiply two 32-bit integer values and add product with value stored in the A10 register of the MMAU returning a 64-bit integer A10 register value.

The MMAU_d_smac_ll function multiplies two 32-bit integer values and add product with value stored in the A10 register of the MMAU returning a 64-bit integer A10 register value.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- lval1 – int32_t integer value.
- lval2 – int32_t integer value.

Returns

int64_t integer value after multiply.

```
static inline int64_t MMAU_d_smacs_ll (register int32_t lval1, register int32_t lval2)
```

Saturating multiply two 32-bit integer values and add product with value stored in the A10 register of the MMAU returning a 64-bit integer A10 register value.

The MMAU_d_smacs_ll function multiplies two 32-bit integer values and add product with value stored in the A10 register of the MMAU returning saturated 64-bit integer A10 register value.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- lval1 – int32_t integer value.
- lval2 – int32_t integer value.

Returns

int64_t integer value after multiply.

```
static inline int64_t MMAU_d_smac_dl (register int64_t dval, register int32_t lval)
```

Multiply 64-bit integer value with 32-bit integer value and add product with value stored in the A10 register of the MMAU returning a 64-bit integer A10 register value.

The MMAU_d_smac_dl function multiplies 64-bit integer value with 32-bit integer value and add product with value stored in the A10 register of the MMAU returning a 64-bit integer A10 register value.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – int64_t integer value.
- lval – int32_t integer value.

Returns

int64_t integer value after multiply.

static inline int64_t MMAU_d_smacs_dl (register int64_t dval, register int32_t lval)

Saturating multiply 64-bit integer value with 32-bit integer value and add product with value stored in the A10 register of the MMAU returning saturated 64-bit integer A10 register value.

The MMAU_d_smacs_dl function multiplies 64-bit integer value with 32-bit integer value and add product with value stored in the A10 register of the MMAU returning saturated 64-bit integer A10 register value.

Note: Saturated product is stored in A10 register of the MMAU for next computation. If saturation occurs, the instruction sets the accumulation overflow (Q) and multiply or divide overflow (V) flags to 1 in the CSR. Otherwise, it clears the Q and V flags.

Parameters

- dval – int64_t integer value.
- lval – int32_t integer value.

Returns

int64_t integer value after multiply.

static inline int64_t MMAU_d_smaca_dl (register int64_t dval, register int32_t lval)

Multiply 32-bit integer value by value stored in the A10 register of the MMAU and add product with 64-bit integer value returning a 64-bit integer A10 register value.

The MMAU_d_smaca_dl function multiplies 32-bit integer value by value stored in the A10 register of the MMAU and add product with 64-bit integer value returning a 64-bit integer A10 register value.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – int64_t integer value.
- lval – int32_t integer value.

Returns

int64_t integer value after multiply.

static inline int64_t MMAU_d_smacas_dl (register int64_t dval, register int32_t lval)

Saturating multiply 32-bit integer value by value stored in the A10 register of the MMAU and add product with 64-bit integer value returning a saturated 64-bit integer A10 register value.

The MMAU_d_smacas_dl function multiplies 32-bit integer value by value stored in the A10 register of the MMAU and add product with 64-bit integer value returning saturated 64-bit integer A10 register value.

Note: Saturated product is stored in A10 register of the MMAU for next computation. If saturation occurs, the instruction sets the accumulation overflow (Q) and multiply or divide overflow (V) flags to 1 in the CSR. Otherwise, it clears the Q and V flags.

Parameters

- dval – int64_t integer value.
- lval – int32_t integer value.

Returns

int64_t integer value after multiply.

static inline int32_t MMAU_l_sdiv_ll (register int32_t lnum, register int32_t lden)

Divide two 32-bit integer values returning a 32-bit integer quotient.

The MMAU_l_sdiv_ll function divides two 32-bit integer values returning a 32-bit integer quotient.

Note: Quotient is stored in A0 register of the MMAU for next computation.

Parameters

- lnum – int32_t integer divisor value.
- lden – int32_t integer dividend value.

Returns

int32_t integer quotient value.

static inline int32_t MMAU_l_sdivs_ll (register int32_t lnum, register int32_t lden)

Divide two 32-bit integer values returning a 32-bit integer quotient.

The MMAU_l_sdivs_ll function divides two 32-bit integer values returning a 32-bit integer quotient.

Note: Saturated quotient is stored in A0 register of the MMAU for next computation.

Parameters

- lnum – int32_t integer divisor value.
- lden – int32_t integer dividend value.

Returns

int32_t integer quotient value.

static inline int64_t MMAU_d_sdiv_dl (register int64_t dnum, register int32_t lden)

Divide 64-bit integer value by 32-bit integer value returning a 64-bit integer quotient.

The MMAU_d_sdiv_dl function divides 64-bit integer value by 32-bit integer value returning a 64-bit integer quotient.

Note: Quotient is stored in A10 register of the MMAU for next computation.

Parameters

- dnum – int64_t integer divisor value.
- lden – int32_t integer dividend value.

Returns

int64_t integer quotient value.

static inline int64_t MMAU_d_sdivs_dl (register int64_t dnum, register int32_t lden)

Divide 64-bit integer value by 32-bit integer value returning a 64-bit integer quotient.

The MMAU_d_sdivs_dl function divides 64-bit integer value by 32-bit integer value returning a 64-bit integer quotient.

Note: Saturated quotient is stored in A10 register of the MMAU for next computation.

Parameters

- `dnum` – `int64_t` integer divisor value.
- `lden` – `int32_t` integer dividend value.

Returns

`int64_t` integer quotient value.

`static inline int64_t MMAU_d_sdiv_dd (register int64_t dnum, register int64_t dden)`

Divide two 64-bit integer values returning a 64-bit integer quotient.

The `MMAU_d_sdiv_dd` function divides two 64-bit integer values returning a 64-bit integer quotient.

Note: Quotient is stored in A10 register of the MMAU for next computation.

Parameters

- `dnum` – `int64_t` integer divisor value.
- `dden` – `int64_t` integer dividend value.

Returns

`int64_t` integer quotient value.

`static inline int64_t MMAU_d_sdivs_dd (register int64_t dnum, register int64_t dden)`

Divide two 64-bit integer values returning a 64-bit integer quotient.

The `MMAU_d_sdivs_dd` function divides two 64-bit integer values returning a 64-bit integer quotient.

Note: Saturated quotient is stored in A10 register of the MMAU for next computation.

Parameters

- `dnum` – `int64_t` integer divisor value.
- `dden` – `int64_t` integer dividend value.

Returns

`int64_t` integer quotient value.

`static inline int64_t MMAU_d_sdiva_l (register int32_t lden1)`

Divide 32-bit integer value stored in the A10 register of the MMAU by 32-bit integer value returning a 64-bit integer quotient.

The `MMAU_d_sdiva_l` function divides 32-bit integer value stored in the A10 register of the MMAU by 32-bit integer value returning a 64-bit integer quotient.

Note: Quotient is stored in A10 register of the MMAU for next computation.

Parameters

- `lden1` – `int32_t` integer dividend value.

Returns

int64_t integer quotient value.

static inline int64_t MMAU_d_sdivas_l (register int32_t lden1)

Divide 32-bit integer value stored in the A10 register of the MMAU by 32-bit integer value returning saturated 64-bit integer quotient.

The MMAU_d_sdivas_l function divides 32-bit integer value stored in the A10 register of the MMAU by 32-bit integer value returning a saturated 64-bit integer quotient.

Note: Saturated quotient is stored in A10 register of the MMAU for next computation.

Parameters

- lden1 – int32_t integer dividend value.

Returns

int64_t integer quotient value.

static inline int64_t MMAU_d_sdiva_d (register int64_t dden1)

Divide 64-bit integer value stored in the A10 register of the MMAU by 64-bit integer value returning a 64-bit integer quotient.

The MMAU_d_sdiva_d function divides 64-bit integer value stored in the A10 register of the MMAU by 64-bit integer value returning a 64-bit integer quotient.

Note: Quotient is stored in A10 register of the MMAU for next computation.

Parameters

- dden1 – int64_t integer dividend value.

Returns

int64_t integer quotient value.

static inline int64_t MMAU_d_sdivas_d (register int64_t dden1)

Divide 64-bit integer value stored in the A10 register of the MMAU by 64-bit integer value returning saturated 64-bit integer quotient.

The MMAU_d_sdivas_d function divides 64-bit integer value stored in the A10 register of the MMAU by 64-bit integer value returning a saturated 64-bit integer quotient.

Note: Saturated quotient is stored in A10 register of the MMAU for next computation.

Parameters

- dden1 – int64_t integer dividend value.

Returns

int64_t integer quotient value.

static inline void MMAU_lda_d (register frac64_t dval)

Load A10 accumulator register of the MMAU by 64-bit fractional value.

The MMAU_lda_d function loads A10 accumulator register of the MMAU by 64-bit fractional value.

Parameters

- dval – frac64_t fractional value.

static inline frac32_t MMAU_1_mul_ll (register frac32_t lval1, register frac32_t lval2)

Multiply two 32-bit fractional values returning a 32-bit fractional product.

The MMAU_1_mul_ll function multiplies two 32-bit fractional values returning a 32-bit fractional product.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- lval1 – frac32_t fractional value.
- lval2 – frac32_t fractional value.

Returns

frac32_t fractional value after multiply.

static inline frac32_t MMAU_1_muls_ll (register frac32_t lval1, register frac32_t lval2)

Saturating multiply two 32-bit fractional values returning saturated 32-bit fractional product.

The MMAU_1_muls_ll function multiplies two 32-bit fractional values returning saturated 32-bit fractional product.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- lval1 – frac32_t fractional value.
- lval2 – frac32_t fractional value.

Returns

frac32_t fractional value after multiply.

static inline frac64_t MMAU_d_mul_ll (register frac32_t lval1, register frac32_t lval2)

Multiply two 32-bit fractional values returning a 64-bit fractional product.

The MMAU_d_mul_ll function multiplies two 32-bit fractional values returning a 64-bit fractional product.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- lval1 – frac32_t fractional value.
- lval2 – frac32_t fractional value.

Returns

frac64_t fractional value after multiply.

static inline frac64_t MMAU_d_muls_ll (register frac32_t lval1, register frac32_t lval2)

Saturating multiply two 32-bit fractional values returning saturated 64-bit fractional product.

The MMAU_d_muls_ll function multiplies two 32-bit fractional values returning saturated 64-bit fractional product.

Note: Saturated product is stored in A10 register of the MMAU for next computation. If saturation occurs, the instruction sets the accumulation overflow (Q) and multiply or divide overflow (V) flags to 1 in the CSR. Otherwise, it clears the Q and V flags.

Parameters

- lval1 – frac32_t fractional value.
- lval2 – frac32_t fractional value.

Returns

frac64_t fractional value after multiply.

static inline frac64_t MMAU_d_mul_dl (register frac64_t dval, register frac32_t lval)

Multiply 64-bit fractional value with 32-bit fractional value returning a 64-bit fractional product.

The MMAU_d_mul_dl function multiplies 64-bit fractional value with 32-bit fractional value returning a 64-bit fractional product.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – frac64_t fractional value.
- lval – frac32_t fractional value.

Returns

frac64_t fractional value after multiply.

static inline frac64_t MMAU_d_muls_dl (register frac64_t dval, register frac32_t lval)

Saturating multiply 64-bit fractional value with 32-bit fractional value returning saturated 64-bit fractional product.

The MMAU_d_muls_dl function multiplies 64-bit fractional value with 32-bit fractional value returning saturated 64-bit fractional product.

Note: Saturated product is stored in A10 register of the MMAU for next computation. If saturation occurs, the instruction sets the accumulation overflow (Q) and multiply or divide overflow (V) flags to 1 in the CSR. Otherwise, it clears the Q and V flags.

Parameters

- dval – frac64_t fractional value.
- lval – frac32_t fractional value.

Returns

frac64_t fractional value after multiply.

static inline frac64_t MMAU_d_mula_l (register frac32_t lval)

Multiply 32-bit fractional value with 64-bit fractional value stored in the A10 register of the MMAU returning a 64-bit fractional product.

The MMAU_d_mula_l function multiplies 32-bit fractional value with 64-bit fractional value stored in the A10 register of the MMAU returning a 64-bit fractional product.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- lval – frac32_t fractional value.

Returns

frac64_t fractional value after multiply.

static inline frac64_t MMAU_d_mulas_l (register frac32_t lval)

Saturating multiply 32-bit fractional value with 64-bit fractional value stored in the A10 register of the MMAU returning saturated 64-bit fractional product.

The MMAU_d_mulas_l function multiplies 32-bit fractional value with 64-bit fractional value stored in the A10 register of the MMAU returning saturated 64-bit fractional product.

Note: Saturated product is stored in A10 register of the MMAU for next computation. If saturation occurs, the instruction sets the accumulation overflow (Q) and multiply or divide overflow (V) flags to 1 in the CSR. Otherwise, it clears the Q and V flags.

Parameters

- lval – frac32_t fractional value.

Returns

frac64_t fractional value after multiply.

static inline frac32_t MMAU_l_mul_dl (register frac64_t dval, register frac32_t lval)

Multiply 64-bit fractional value with 32-bit fractional value returning a 32-bit fractional product.

The MMAU_l_mul_dl function multiplies 64-bit fractional value with 32-bit fractional value returning a 32-bit fractional product.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – frac64_t fractional value.
- lval – frac32_t fractional value.

Returns

frac32_t fractional value after multiply.

static inline frac32_t MMAU_l_muls_dl (register frac64_t dval, register frac32_t lval)

Saturating multiply 64-bit fractional value with 32-bit fractional value returning saturated 32-bit fractional product.

The MMAU_l_muls_dl function multiplies 64-bit fractional value with 32-bit fractional value returning saturated 32-bit fractional product.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – frac64_t fractional value.
- lval – frac32_t fractional value.

Returns

frac32_t fractional value after multiply.

static inline frac32_t MMAU_l_mula_l (register frac32_t lval)

Multiply 32-bit fractional value with 64-bit fractional value stored in the A10 register of the MMAU returning a 32-bit fractional product.

The MMAU_l_mula_l function multiplies 32-bit fractional value with 64-bit fractional value stored in the A10 register of the MMAU returning a 32-bit fractional product.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- lval – frac32_t fractional value.

Returns

frac32_t fractional value after multiply.

static inline frac32_t MMAU_l_mulas_l (register frac32_t lval)

Saturating multiply 32-bit fractional value with 64-bit fractional value stored in the A10 register of the MMAU returning saturated 32-bit fractional product.

The MMAU_l_mulas_l function multiplies 32-bit fractional value with 64-bit fractional value stored in the A10 register of the MMAU returning saturated 32-bit fractional product.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- lval – frac32_t fractional value.

Returns

frac32_t fractional value after multiply.

static inline frac64_t MMAU_d_mac_ll (register frac32_t lval1, register frac32_t lval2)

Multiply two 32-bit fractional values and add product with value stored in the A10 register of the MMAU returning a 64-bit fractional A10 register value.

The MMAU_d_mac_ll function multiplies two 32-bit fractional values and add product with value stored in the A10 register of the MMAU returning a 64-bit fractional A10 register value.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- lval1 – frac32_t fractional value.
- lval2 – frac32_t fractional value.

Returns

frac64_t fractional value after multiply.

static inline frac64_t MMAU_d_macs_ll (register frac32_t lval1, register frac32_t lval2)

Saturating multiply two 32-bit fractional values and add product with value stored in the A10 register of the MMAU returning a 64-bit fractional A10 register value.

The MMAU_d_macs_ll function multiplies two 32-bit fractional values and add product with value stored in the A10 register of the MMAU returning saturated 64-bit fractional A10 register value.

Note: Saturated product is stored in A10 register of the MMAU for next computation. If saturation occurs, the instruction sets the accumulation overflow (Q) and multiply or divide overflow (V) flags to 1 in the CSR. Otherwise, it clears the Q and V flags.

Parameters

- lval1 – frac32_t fractional value.
- lval2 – frac32_t fractional value.

Returns

frac64_t fractional value after multiply.

static inline frac64_t MMAU_d_mac_dl (register frac64_t dval, register frac32_t lval)

Multiply 64-bit fractional value with 32-bit fractional value and add product with value stored in the A10 register of the MMAU returning a 64-bit fractional A10 register value.

The MMAU_d_mac_dl function multiplies 64-bit fractional value with 32-bit fractional value and add product with value stored in the A10 register of the MMAU returning a 64-bit fractional A10 register value.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – frac64_t fractional value.
- lval – frac32_t fractional value.

Returns

frac64_t fractional value after multiply.

static inline frac64_t MMAU_d_macs_dl (register frac64_t dval, register frac32_t lval)

Saturating multiply 64-bit fractional value with 32-bit fractional value and add product with value stored in the A10 register of the MMAU returning saturated 64-bit fractional A10 register value.

The MMAU_d_macs_dl function multiplies 64-bit fractional value with 32-bit fractional value and add product with value stored in the A10 register of the MMAU returning saturated 64-bit fractional A10 register value.

Note: Saturated product is stored in A10 register of the MMAU for next computation. If saturation occurs, the instruction sets the accumulation overflow (Q) and multiply or divide overflow (V) flags to 1 in the CSR. Otherwise, it clears the Q and V flags.

Parameters

- dval – frac64_t fractional value.
- lval – frac32_t fractional value.

Returns

frac64_t fractional value after multiply.

static inline frac64_t MMAU_d_maca_dl (register frac64_t dval, register frac32_t lval)

Multiply 32-bit fractional value by value stored in the A10 register of the MMAU and add product with 64-bit fractional value returning a 64-bit fractional A10 register value.

The MMAU_d_macas_dl function multiplies 32-bit fractional value by value stored in the A10 register of the MMAU and add product with 64-bit fractional value returning a 64-bit fractional A10 register value.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- dval – frac64_t fractional value.
- lval – frac32_t fractional value.

Returns

frac64_t fractional value after multiply.

```
static inline frac64_t MMAU_d_macas_dl (register frac64_t dval, register frac32_t lval)
```

Saturating multiply 32-bit fractional value by value stored in the A10 register of the MMAU and add product with 64-bit fractional value returning a saturated 64-bit fractional A10 register value.

The MMAU_d_macas_dl function multiplies 32-bit fractional value by value stored in the A10 register of the MMAU and add product with 64-bit fractional value returning saturated 64-bit fractional A10 register value.

Note: Saturated product is stored in A10 register of the MMAU for next computation. If saturation occurs, the instruction sets the accumulation overflow (Q) and multiply or divide overflow (V) flags to 1 in the CSR. Otherwise, it clears the Q and V flags.

Parameters

- dval – frac64_t fractional value.
- lval – frac32_t fractional value.

Returns

frac64_t fractional value after multiply.

```
static inline frac32_t MMAU_l_mac_ll (register frac32_t lval1, register frac32_t lval2)
```

Multiply two 32-bit fractional values and add product with value stored in the A10 register of the MMAU returning a 32-bit fractional A10 register value.

The MMAU_l_mac_ll function multiplies two 32-bit fractional values and add product with value stored in the A10 register of the MMAU returning a 32-bit fractional A1 register value.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- lval1 – frac32_t fractional value.
- lval2 – frac32_t fractional value.

Returns

frac32_t fractional value after multiply.

```
static inline frac32_t MMAU_l_mac_ll (register frac32_t lval1, register frac32_t lval2)
```

Saturating multiply two 32-bit fractional values and add product with value stored in the A10 register of the MMAU returning a 32-bit fractional A10 register value.

The `MMAU_1_macs_dl` function multiplies two 32-bit fractional values and add product with value stored in the A10 register of the MMAU returning saturated 32-bit fractional A1 register value.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- `lval1` – `frac32_t` fractional value.
- `lval2` – `frac32_t` fractional value.

Returns

`frac32_t` fractional value after multiply.

```
static inline frac32_t MMAU_1_mac_dl (register frac64_t dval, register frac32_t lval)
```

Multiply 64-bit fractional value with 32-bit fractional value and add product with value stored in the A10 register of the MMAU returning a 32-bit fractional A10 register value.

The `MMAU_1_mac_dl` function multiplies 64-bit fractional value with 32-bit fractional value and add product with value stored in the A10 register of the MMAU returning a 32-bit fractional A1 register value.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- `dval` – `frac64_t` fractional value.
- `lval` – `frac32_t` fractional value.

Returns

`frac32_t` fractional value after multiply.

```
static inline frac32_t MMAU_1_macs_dl (register frac64_t dval, register frac32_t lval)
```

Saturating multiply 64-bit fractional value with 32-bit fractional value and add product with value stored in the A10 register of the MMAU returning saturated 32-bit fractional A10 register value.

The `MMAU_1_macs_dl` function multiplies 64-bit fractional value with 32-bit fractional value and add product with value stored in the A10 register of the MMAU returning saturated 32-bit fractional A1 register value.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- `dval` – `frac64_t` fractional value.
- `lval` – `frac32_t` fractional value.

Returns

`frac32_t` fractional value after multiply.

```
static inline frac32_t MMAU_1_maca_dl (register frac64_t dval, register frac32_t lval)
```

Multiply 32-bit fractional value by value stored in the A10 register of the MMAU and add product with 64-bit fractional value returning a 32-bit fractional A10 register value.

The `MMAU_1_macas_dl` function multiplies 32-bit fractional value by value stored in the A10 register of the MMAU and add product with 64-bit fractional value returning a 32-bit fractional A1 register value.

Note: Product is stored in A10 register of the MMAU for next computation.

Parameters

- `dval` – `frac64_t` fractional value.
- `lval` – `frac32_t` fractional value.

Returns

`frac32_t` fractional value after multiply.

`static inline frac32_t MMAU_1_macas_dl (register frac64_t dval, register frac32_t lval)`

Saturating multiply 32-bit fractional value by value stored in the A10 register of the MMAU and add product with 64-bit fractional value returning a saturated 32-bit fractional A10 register value.

The `MMAU_1_macas_dl` function multiplies 32-bit fractional value by value stored in the A10 register of the MMAU and add product with 64-bit fractional value returning saturated 32-bit fractional A1 register value.

Note: Saturated product is stored in A10 register of the MMAU for next computation.

Parameters

- `dval` – `frac64_t` fractional value.
- `lval` – `frac32_t` fractional value.

Returns

`frac32_t` fractional value after multiply.

`static inline frac32_t MMAU_1_div_ll (register frac32_t lnum, register frac32_t lden)`

Divide two 32-bit fractional values returning a 32-bit fractional quotient.

The `MMAU_1_div_ll` function divides two 32-bit fractional values returning a 32-bit fractional quotient.

Note: Quotient is stored in A1 register of the MMAU for next computation.

Parameters

- `lnum` – `frac32_t` fractional divisor value.
- `lden` – `frac32_t` fractional dividend value.

Returns

`frac32_t` fractional quotient value.

`static inline frac32_t MMAU_1_divs_ll (register frac32_t lnum, register frac32_t lden)`

Divide two 32-bit fractional values returning a 32-bit fractional quotient.

The `MMAU_1_divs_ll` function divides two 32-bit fractional values returning a 32-bit fractional quotient.

Note: Saturated quotient is stored in A1 register of the MMAU for next computation. If saturation occurs, the instruction sets the accumulation overflow (Q) and multiply or divide overflow (V) flags to 1 in the CSR. Otherwise, it clears the Q and V flags.

Parameters

- lnum – frac32_t fractional divisor value.
- lden – frac32_t fractional dividend value.

Returns

frac32_t fractional quotient value.

static inline frac32_t MMAU_l_divas_l (register frac32_t lden)

Divide 64-bit fractional value stored in the A10 register of the MMAU by 32-bit fractional value returning saturated 32-bit fractional quotient.

The MMAU_l_divas_l function divides 64-bit fractional value stored in the A10 register of the MMAU by 32-bit fractional value returning a saturated 32-bit fractional quotient.

Note: Saturated quotient is stored in A1 register of the MMAU for next computation.

Parameters

- lden – frac32_t fractional dividend value.

Returns

frac32_t fractional quotient value.

static inline frac64_t MMAU_d_div_dl (register frac64_t dnum, register frac32_t lden)

Divide 64-bit fractional value by 32-bit fractional value returning a 64-bit fractional quotient.

The MMAU_d_div_dl function divides 64-bit fractional value by 32-bit fractional value returning a 64-bit fractional quotient.

Note: Quotient is stored in A10 register of the MMAU for next computation.

Parameters

- dnum – frac64_t fractional divisor value.
- lden – frac32_t fractional dividend value.

Returns

frac64_t fractional quotient value.

static inline frac64_t MMAU_d_divs_dl (register frac64_t dnum, register frac32_t lden)

Divide 64-bit fractional value by 32-bit fractional value returning a 64-bit fractional quotient.

The MMAU_d_divs_dl function divides 64-bit fractional value by 32-bit fractional value returning a 64-bit fractional quotient.

Note: Saturated quotient is stored in A10 register of the MMAU for next computation. If saturation occurs, the instruction sets the accumulation overflow (Q) and multiply or divide overflow (V) flags to 1 in the CSR. Otherwise, it clears the Q and V flags.

Parameters

- `dnum` – `frac64_t` fractional divisor value.
- `lden` – `frac32_t` fractional dividend value.

Returns

`frac64_t` fractional quotient value.

static inline `frac64_t` MMAU_d_diva_l (register `frac32_t` `lden1`)

Divide 32-bit fractional value stored in the A10 register of the MMAU by 32-bit fractional value returning a 64-bit fractional quotient.

The MMAU_d_diva_l function divides 32-bit fractional value stored in the A10 register of the MMAU by 32-bit fractional value returning a 64-bit fractional quotient.

Note: Quotient is stored in A10 register of the MMAU for next computation.

Parameters

- `lden1` – `frac32_t` fractional dividend value.

Returns

`frac64_t` fractional quotient value.

static inline `frac64_t` MMAU_d_divas_l (register `frac32_t` `lden1`)

Divide 32-bit fractional value stored in the A10 register of the MMAU by 32-bit fractional value returning saturated 64-bit fractional quotient.

The MMAU_d_divas_l function divides 32-bit fractional value stored in the A10 register of the MMAU by 32-bit fractional value returning a saturated 64-bit fractional quotient.

Note: Saturated quotient is stored in A10 register of the MMAU for next computation. If saturation occurs, the instruction sets the accumulation overflow (Q) and multiply or divide overflow (V) flags to 1 in the CSR. Otherwise, it clears the Q and V flags.

Parameters

- `lden1` – `frac32_t` fractional dividend value.

Returns

`frac64_t` fractional quotient value.

static inline `frac32_t` MMAU_l_diva_l (register `frac32_t` `lden`)

Divide 64-bit fractional value stored in the A10 register of the MMAU by 32-bit fractional value returning a 32-bit fractional quotient.

The MMAU_l_diva_l function divides 64-bit fractional value stored in the A10 register of the MMAU by 32-bit fractional value returning a 32-bit fractional quotient.

Note: Quotient is stored in A1 register of the MMAU for next computation.

Parameters

- `lden` – `frac32_t` fractional dividend value.

Returns

`frac32_t` fractional quotient value.

static inline frac32_t MMAU_1_sqr_l (register frac32_t lrad)

Compute and return a 32-bit fractional square root of the 32-bit fractional radicand.

The MMAU_1_sqr_l function computes and returns a 32-bit fractional square root of the 32-bit fractional radicand.

Note: Square root is stored in A1 register of the MMAU for next computation.

Parameters

- lrad – frac32_t fractional radicand.

Returns

frac32_t fractional square root.

static inline frac32_t MMAU_1_sqr_d (register frac64_t drad)

Compute and return a 32-bit fractional square root of the 64-bit fractional radicand.

The MMAU_1_sqr_d function computes and returns a 32-bit fractional square root of the 64-bit fractional radicand.

Note: Quotient is stored in A1 register of the MMAU for next computation.

Parameters

- drad – frac64_t fractional radicand.

Returns

frac32_t fractional square root.

static inline *frac32_t* MMAU_1_sqra(void)

Compute and return a 32-bit fractional square root of the radicand stored in the A10 register of the MMAU.

The MMAU_1_sqra function computes and returns a 32-bit fractional square root of the radicand stored in the A10 register of the MMAU.

Note: Quotient is stored in A1 register of the MMAU for next computation.

Returns

frac32_t fractional square root.

FSL_MMAU_DRIVER_VERSION

Version 2.0.1

MMAU__X0

Accumulator register X0

MMAU__X1

Accumulator register X1

MMAU__X2

Accumulator register X2

MMAU__X3

Accumulator register X3

MMAU__A0

Accumulator register A0

MMAU__A1

Accumulator register A1

MMAU__A10

Accumulator register pair A10

MMAU__REGRW

Registers RW

MMAU__UMUL

$A10=X2*X3$

MMAU__UMULD

$A10=X21*X3$

MMAU__UMULDA

$A10=A10*X3$

MMAU__UMAC

$A10=X2*X3+A10$

MMAU__UMACD

$A10=X21*X3+A10$

MMAU__UMACDA

$A10=A10*X3+X21$

MMAU__UDIV

$X21/X3=A10$

MMAU__UDIVD

$A10=X2/X3$

MMAU__UDIVDA

$A10=X21/X3$

MMAU__UDIVDD

$A10=A10/X3$

MMAU__UDIVDDA

$A10=A10/X32$

MMAU__USQR

$A10=SQR(X3)$

MMAU__USQRD

$A10=SQR(X32)$

MMAU__USQRDA

$A10=SQR(A10)$

MMAU__QSQR

$A10=SQR(X3)$

MMAU__QSQRD

$A10=SQR(X32)$

MMAU__QSQRDA

$A10=SQR(A10)$

MMAU__QDIV
 $A10=X2/X3$
 MMAU__QDIVD
 $A10=X21/X3$
 MMAU__QDIVDA
 $A10=A10/X3$
 MMAU__QMUL
 $A10=X2*X3$
 MMAU__QMULD
 $A10=X21*X3$
 MMAU__QMULDA
 $A10=A10*X3$
 MMAU__QMAC
 $A10=X2*X3+A10$
 MMAU__QMACD
 $A10=X21*X3+A10$
 MMAU__QMACDA
 $A10=A10*X3+X21$
 MMAU__SMUL
 $A10=X2*X3$
 MMAU__SMULD
 $A10=X21*X3$
 MMAU__SMULDA
 $A10=A10*X3$
 MMAU__SMAC
 $A10=X2*X3+A10$
 MMAU__SMACD
 $A10=X21*X3+A10$
 MMAU__SMACDA
 $A10=A10*X3+X21$
 MMAU__SDIV
 $A10=X2/X3$
 MMAU__SDIVD
 $A10=X21/X3$
 MMAU__SDIVDA
 $A10=A10/X3$
 MMAU__SDIVDD
 $A10=X10/X32$
 MMAU__SDIVDDA
 $A10=A10/X32$
 MMAU__SAT
 Saturation

enum `_mmau_interrupt_enable`

MMAU interrupt configuration structure, default settings all disabled.

This structure contains the settings for all of the MMAU interrupt configurations.

Values:

enumerator `kMMAU__AccumOverflowInterruptEnable`

Accumulation Overflow Enable

enumerator `kMMAU__OverflowInterruptEnable`

Multiply or Divide overflow Enable

enumerator `kMMAU__DivideByZeroInterruptEnable`

Divide by Zero Enable

enum `_mmau_interrupt_flag`

MMAU interrupt and instruction flags.

Values:

enumerator `kMMAU__AccumOverflowInterruptFlag`

Accumulation Overflow Interrupt Flag

enumerator `kMMAU__OverflowInterruptFlag`

Multiply or Divide overflow Interrupt Flag

enumerator `kMMAU__DivideByZeroInterruptFlag`

Divide by Zero Interrupt Flag

enum `_mmau_instruction_flag`

MMAU interrupt and instruction flags.

Values:

enumerator `kMMAU__AccumOverflowInstructionFlag`

Accumulation Overflow

enumerator `kMMAU__OverflowInstructionFlag`

Multiply or Divide overflow

enumerator `kMMAU__DivideByZeroInstructionFlag`

Divide by Zero

enumerator `kMMAU__NegativeInstructionFlag`

Signed calculation result is negative

typedef enum `_mmau_interrupt_enable` `mmau_interrupt_enable_t`

MMAU interrupt configuration structure, default settings all disabled.

This structure contains the settings for all of the MMAU interrupt configurations.

typedef enum `_mmau_interrupt_flag` `mmau_interrupt_flag_t`

MMAU interrupt and instruction flags.

typedef enum `_mmau_instruction_flag` `mmau_instruction_flag_t`

MMAU interrupt and instruction flags.

static inline uint32_t `MMAU_GetHwRevCmd(MMAU_Type *base)`

Gets hardware revision level.

This function gets the hardware revision level of the MMAU. It returns HDR field of the control/status register.

Parameters

- base – MMAU peripheral address.

Returns

uint32_t hardware revision level.

2.33 PDB: Programmable Delay Block

void PDB_Init(PDB_Type *base, const *pdb_config_t* *config)

Initializes the PDB module.

This function initializes the PDB module. The operations included are as follows.

- Enable the clock for PDB instance.
- Configure the PDB module.
- Enable the PDB module.

Parameters

- base – PDB peripheral base address.
- config – Pointer to the configuration structure. See “*pdb_config_t*”.

void PDB_Deinit(PDB_Type *base)

De-initializes the PDB module.

Parameters

- base – PDB peripheral base address.

void PDB_GetDefaultConfig(*pdb_config_t* *config)

Initializes the PDB user configuration structure.

This function initializes the user configuration structure to a default value. The default values are as follows.

```
config->loadValueMode = kPDB_LoadValueImmediately;
config->prescalerDivider = kPDB_PrescalerDivider1;
config->dividerMultiplicationFactor = kPDB_DividerMultiplicationFactor1;
config->triggerInputSource = kPDB_TriggerSoftware;
config->enableContinuousMode = false;
```

Parameters

- config – Pointer to configuration structure. See “*pdb_config_t*”.

static inline void PDB_Enable(PDB_Type *base, bool enable)

Enables the PDB module.

Parameters

- base – PDB peripheral base address.
- enable – Enable the module or not.

static inline void PDB_DoSoftwareTrigger(PDB_Type *base)

Triggers the PDB counter by software.

Parameters

- base – PDB peripheral base address.

static inline void PDB_DoLoadValues(PDB_Type *base)

Loads the counter values.

This function loads the counter values from the internal buffer. See “pdb_load_value_mode_t” about PDB’s load mode.

Parameters

- base – PDB peripheral base address.

static inline void PDB_EnableDMA(PDB_Type *base, bool enable)

Enables the DMA for the PDB module.

Parameters

- base – PDB peripheral base address.
- enable – Enable the feature or not.

static inline void PDB_EnableInterrupts(PDB_Type *base, uint32_t mask)

Enables the interrupts for the PDB module.

Parameters

- base – PDB peripheral base address.
- mask – Mask value for interrupts. See “_pdb_interrupt_enable”.

static inline void PDB_DisableInterrupts(PDB_Type *base, uint32_t mask)

Disables the interrupts for the PDB module.

Parameters

- base – PDB peripheral base address.
- mask – Mask value for interrupts. See “_pdb_interrupt_enable”.

static inline uint32_t PDB_GetStatusFlags(PDB_Type *base)

Gets the status flags of the PDB module.

Parameters

- base – PDB peripheral base address.

Returns

Mask value for asserted flags. See “_pdb_status_flags”.

static inline void PDB_ClearStatusFlags(PDB_Type *base, uint32_t mask)

Clears the status flags of the PDB module.

Parameters

- base – PDB peripheral base address.
- mask – Mask value of flags. See “_pdb_status_flags”.

static inline void PDB_SetModulusValue(PDB_Type *base, uint32_t value)

Specifies the counter period.

Parameters

- base – PDB peripheral base address.
- value – Setting value for the modulus. 16-bit is available.

static inline uint32_t PDB_GetCounterValue(PDB_Type *base)

Gets the PDB counter’s current value.

Parameters

- base – PDB peripheral base address.

Returns

PDB counter's current value.

static inline void PDB_SetCounterDelayValue(PDB_Type *base, uint32_t value)

Sets the value for the PDB counter delay event.

Parameters

- base – PDB peripheral base address.
- value – Setting value for PDB counter delay event. 16-bit is available.

static inline void PDB_SetADCPreTriggerConfig(PDB_Type *base, *pdb_adc_trigger_channel_t* channel, *pdb_adc_pretrigger_config_t* *config)

Configures the ADC pre-trigger in the PDB module.

Parameters

- base – PDB peripheral base address.
- channel – Channel index for ADC instance.
- config – Pointer to the configuration structure. See “*pdb_adc_pretrigger_config_t*”.

static inline void PDB_SetADCPreTriggerDelayValue(PDB_Type *base, *pdb_adc_trigger_channel_t* channel, *pdb_adc_pretrigger_t* pretriggerNumber, uint32_t value)

Sets the value for the ADC pre-trigger delay event.

This function sets the value for ADC pre-trigger delay event. It specifies the delay value for the channel's corresponding pre-trigger. The pre-trigger asserts when the PDB counter is equal to the set value.

Parameters

- base – PDB peripheral base address.
- channel – Channel index for ADC instance.
- pretriggerNumber – Channel group index for ADC instance.
- value – Setting value for ADC pre-trigger delay event. 16-bit is available.

static inline uint32_t PDB_GetADCPreTriggerStatusFlags(PDB_Type *base, *pdb_adc_trigger_channel_t* channel)

Gets the ADC pre-trigger's status flags.

Parameters

- base – PDB peripheral base address.
- channel – Channel index for ADC instance.

Returns

Mask value for asserted flags. See “*_pdb_adc_pretrigger_flags*”.

static inline void PDB_ClearADCPreTriggerStatusFlags(PDB_Type *base, *pdb_adc_trigger_channel_t* channel, uint32_t mask)

Clears the ADC pre-trigger status flags.

Parameters

- base – PDB peripheral base address.
- channel – Channel index for ADC instance.
- mask – Mask value for flags. See “*_pdb_adc_pretrigger_flags*”.

```
void PDB_SetDACTriggerConfig(PDB_Type *base, pdb_dac_trigger_channel_t channel,
                             pdb_dac_trigger_config_t *config)
```

Configures the DAC trigger in the PDB module.

Parameters

- base – PDB peripheral base address.
- channel – Channel index for DAC instance.
- config – Pointer to the configuration structure. See “*pdb_dac_trigger_config_t*”.

```
static inline void PDB_SetDACTriggerIntervalValue(PDB_Type *base, pdb_dac_trigger_channel_t
                                                    channel, uint32_t value)
```

Sets the value for the DAC interval event.

This function sets the value for DAC interval event. DAC interval trigger triggers the DAC module to update the buffer when the DAC interval counter is equal to the set value.

Parameters

- base – PDB peripheral base address.
- channel – Channel index for DAC instance.
- value – Setting value for the DAC interval event.

```
static inline void PDB_EnablePulseOutTrigger(PDB_Type *base, pdb_pulse_out_channel_mask_t
                                              channelMask, bool enable)
```

Enables the pulse out trigger channels.

Parameters

- base – PDB peripheral base address.
- channelMask – Channel mask value for multiple pulse out trigger channel.
- enable – Whether the feature is enabled or not.

```
static inline void PDB_SetPulseOutTriggerDelayValue(PDB_Type *base,
                                                     pdb_pulse_out_trigger_channel_t channel,
                                                     uint32_t value1, uint32_t value2)
```

Sets event values for the pulse out trigger.

This function is used to set event values for the pulse output trigger. These pulse output trigger delay values specify the delay for the PDB Pulse-out. Pulse-out goes high when the PDB counter is equal to the pulse output high value (value1). Pulse-out goes low when the PDB counter is equal to the pulse output low value (value2).

Parameters

- base – PDB peripheral base address.
- channel – Channel index for pulse out trigger channel.
- value1 – Setting value for pulse out high.
- value2 – Setting value for pulse out low.

FSL_PDB_DRIVER_VERSION

PDB driver version 2.0.4.

```
enum __pdb_status_flags
```

PDB flags.

Values:

enumerator kPDB_LoadOKFlag

This flag is automatically cleared when the values in buffers are loaded into the internal registers after the LDOK bit is set or the PDBEN is cleared.

enumerator kPDB_DelayEventFlag

PDB timer delay event flag.

enum __pdb_adc_pretrigger_flags

PDB ADC PreTrigger channel flags.

Values:

enumerator kPDB_ADPreTriggerChannel0Flag

Pre-trigger 0 flag.

enumerator kPDB_ADPreTriggerChannel1Flag

Pre-trigger 1 flag.

enumerator kPDB_ADPreTriggerChannel0ErrorFlag

Pre-trigger 0 Error.

enumerator kPDB_ADPreTriggerChannel1ErrorFlag

Pre-trigger 1 Error.

enum __pdb_interrupt_enable

PDB buffer interrupts.

Values:

enumerator kPDB_SequenceErrorInterruptEnable

PDB sequence error interrupt enable.

enumerator kPDB_DelayInterruptEnable

PDB delay interrupt enable.

enum __pdb_load_value_mode

PDB load value mode.

Selects the mode to load the internal values after doing the load operation (write 1 to PDBx_SC[LDOK]). These values are for the following operations.

- PDB counter (PDBx_MOD, PDBx_IDLY)
- ADC trigger (PDBx_CHnDLYm)
- DAC trigger (PDBx_DACINTx)
- CMP trigger (PDBx_POyDLY)

Values:

enumerator kPDB_LoadValueImmediately

Load immediately after 1 is written to LDOK.

enumerator kPDB_LoadValueOnCounterOverflow

Load when the PDB counter overflows (reaches the MOD register value).

enumerator kPDB_LoadValueOnTriggerInput

Load a trigger input event is detected.

enumerator kPDB_LoadValueOnCounterOverflowOrTriggerInput

Load either when the PDB counter overflows or a trigger input is detected.

enum __pdb_prescaler_divider

Prescaler divider.

Counting uses the peripheral clock divided by multiplication factor selected by times of MULT.

Values:

enumerator kPDB_PrescalerDivider1

Divider x1.

enumerator kPDB_PrescalerDivider2

Divider x2.

enumerator kPDB_PrescalerDivider4

Divider x4.

enumerator kPDB_PrescalerDivider8

Divider x8.

enumerator kPDB_PrescalerDivider16

Divider x16.

enumerator kPDB_PrescalerDivider32

Divider x32.

enumerator kPDB_PrescalerDivider64

Divider x64.

enumerator kPDB_PrescalerDivider128

Divider x128.

enum __pdb_divider_multiplication_factor

Multiplication factor select for prescaler.

Selects the multiplication factor of the prescaler divider for the counter clock.

Values:

enumerator kPDB_DividerMultiplicationFactor1

Multiplication factor is 1.

enumerator kPDB_DividerMultiplicationFactor10

Multiplication factor is 10.

enumerator kPDB_DividerMultiplicationFactor20

Multiplication factor is 20.

enumerator kPDB_DividerMultiplicationFactor40

Multiplication factor is 40.

enum __pdb_trigger_input_source

Trigger input source.

Selects the trigger input source for the PDB. The trigger input source can be internal or external (EXTRG pin), or the software trigger. See chip configuration details for the actual PDB input trigger connections.

Values:

enumerator kPDB_TriggerInput0

Trigger-In 0.

enumerator kPDB_TriggerInput1

Trigger-In 1.

enumerator kPDB_TriggerInput2

Trigger-In 2.

enumerator kPDB_TriggerInput3

Trigger-In 3.

enumerator kPDB_TriggerInput4

Trigger-In 4.

enumerator kPDB_TriggerInput5

Trigger-In 5.

enumerator kPDB_TriggerInput6

Trigger-In 6.

enumerator kPDB_TriggerInput7

Trigger-In 7.

enumerator kPDB_TriggerInput8

Trigger-In 8.

enumerator kPDB_TriggerInput9

Trigger-In 9.

enumerator kPDB_TriggerInput10

Trigger-In 10.

enumerator kPDB_TriggerInput11

Trigger-In 11.

enumerator kPDB_TriggerInput12

Trigger-In 12.

enumerator kPDB_TriggerInput13

Trigger-In 13.

enumerator kPDB_TriggerInput14

Trigger-In 14.

enumerator kPDB_TriggerSoftware

Trigger-In 15, software trigger.

enum _pdb_adc_trigger_channel

List of PDB ADC trigger channels.

Note: Actual number of available channels is SoC dependent

Values:

enumerator kPDB_ADCTriggerChannel0

PDB ADC trigger channel number 0

enumerator kPDB_ADCTriggerChannel1

PDB ADC trigger channel number 1

enumerator kPDB_ADCTriggerChannel2

PDB ADC trigger channel number 2

enumerator kPDB_ADCTriggerChannel3

PDB ADC trigger channel number 3

enum _pdb_adc_pretrigger

List of PDB ADC pretrigger.

Note: Actual number of available pretrigger channels is SoC dependent

Values:

enumerator kPDB_ADPreTrigger0

PDB ADC pretrigger number 0

enumerator kPDB_ADPreTrigger1

PDB ADC pretrigger number 1

enumerator kPDB_ADPreTrigger2

PDB ADC pretrigger number 2

enumerator kPDB_ADPreTrigger3

PDB ADC pretrigger number 3

enumerator kPDB_ADPreTrigger4

PDB ADC pretrigger number 4

enumerator kPDB_ADPreTrigger5

PDB ADC pretrigger number 5

enumerator kPDB_ADPreTrigger6

PDB ADC pretrigger number 6

enumerator kPDB_ADPreTrigger7

PDB ADC pretrigger number 7

enum _pdb_dac_trigger_channel

List of PDB DAC trigger channels.

Note: Actual number of available channels is SoC dependent

Values:

enumerator kPDB_DACTriggerChannel0

PDB DAC trigger channel number 0

enumerator kPDB_DACTriggerChannel1

PDB DAC trigger channel number 1

enum _pdb_pulse_out_trigger_channel

List of PDB pulse out trigger channels.

Note: Actual number of available channels is SoC dependent

Values:

enumerator kPDB_PulseOutTriggerChannel0

PDB pulse out trigger channel number 0

enumerator kPDB_PulseOutTriggerChannel1

PDB pulse out trigger channel number 1

enumerator kPDB_PulseOutTriggerChannel2
PDB pulse out trigger channel number 2

enumerator kPDB_PulseOutTriggerChannel3
PDB pulse out trigger channel number 3

enum __pdb_pulse_out_channel_mask
List of PDB pulse out trigger channels mask.

Note: Actual number of available channels mask is SoC dependent

Values:

enumerator kPDB_PulseOutChannel0Mask
PDB pulse out trigger channel number 0 mask

enumerator kPDB_PulseOutChannel1Mask
PDB pulse out trigger channel number 1 mask

enumerator kPDB_PulseOutChannel2Mask
PDB pulse out trigger channel number 2 mask

enumerator kPDB_PulseOutChannel3Mask
PDB pulse out trigger channel number 3 mask

typedef enum __pdb_load_value_mode pdb_load_value_mode_t
PDB load value mode.

Selects the mode to load the internal values after doing the load operation (write 1 to PDBx_SC[LDOK]). These values are for the following operations.

- PDB counter (PDBx_MOD, PDBx_IDLY)
- ADC trigger (PDBx_CHnDLYm)
- DAC trigger (PDBx_DACINTx)
- CMP trigger (PDBx_POyDLY)

typedef enum __pdb_prescaler_divider pdb_prescaler_divider_t
Prescaler divider.

Counting uses the peripheral clock divided by multiplication factor selected by times of MULT.

typedef enum __pdb_divider_multiplication_factor pdb_divider_multiplication_factor_t
Multiplication factor select for prescaler.

Selects the multiplication factor of the prescaler divider for the counter clock.

typedef enum __pdb_trigger_input_source pdb_trigger_input_source_t
Trigger input source.

Selects the trigger input source for the PDB. The trigger input source can be internal or external (EXTRG pin), or the software trigger. See chip configuration details for the actual PDB input trigger connections.

typedef enum __pdb_adc_trigger_channel pdb_adc_trigger_channel_t
List of PDB ADC trigger channels.

Note: Actual number of available channels is SoC dependent

```
typedef enum _pdb_adc_pretrigger pdb_adc_pretrigger_t
    List of PDB ADC pretrigger.
```

Note: Actual number of available pretrigger channels is SoC dependent

```
typedef enum _pdb_dac_trigger_channel pdb_dac_trigger_channel_t
    List of PDB DAC trigger channels.
```

Note: Actual number of available channels is SoC dependent

```
typedef enum _pdb_pulse_out_trigger_channel pdb_pulse_out_trigger_channel_t
    List of PDB pulse out trigger channels.
```

Note: Actual number of available channels is SoC dependent

```
typedef enum _pdb_pulse_out_channel_mask pdb_pulse_out_channel_mask_t
    List of PDB pulse out trigger channels mask.
```

Note: Actual number of available channels mask is SoC dependent

```
typedef struct _pdb_config pdb_config_t
    PDB module configuration.
```

```
typedef struct _pdb_adc_pretrigger_config pdb_adc_pretrigger_config_t
    PDB ADC Pre-trigger configuration.
```

```
typedef struct _pdb_dac_trigger_config pdb_dac_trigger_config_t
    PDB DAC trigger configuration.
```

```
struct _pdb_config
    #include <fsl_pdb.h> PDB module configuration.
```

Public Members

pdb_load_value_mode_t loadValueMode
Select the load value mode.

pdb_prescaler_divider_t prescalerDivider
Select the prescaler divider.

pdb_divider_multiplication_factor_t dividerMultiplicationFactor
Multiplication factor select for prescaler.

pdb_trigger_input_source_t triggerInputSource
Select the trigger input source.

bool enableContinuousMode
Enable the PDB operation in Continuous mode.

```
struct _pdb_adc_pretrigger_config
    #include <fsl_pdb.h> PDB ADC Pre-trigger configuration.
```

Public Members

uint32_t enablePreTriggerMask
PDB Channel Pre-trigger Enable.

uint32_t enableOutputMask
PDB Channel Pre-trigger Output Select. PDB channel's corresponding pre-trigger asserts when the counter reaches the channel delay register.

uint32_t enableBackToBackOperationMask
PDB Channel pre-trigger Back-to-Back Operation Enable. Back-to-back operation enables the ADC conversions complete to trigger the next PDB channel pre-trigger and trigger output, so that the ADC conversions can be triggered on next set of configuration and results registers.

struct __pdb_dac_trigger_config
#include <fsl_pdb.h> PDB DAC trigger configuration.

Public Members

bool enableExternalTriggerInput
Enables the external trigger for DAC interval counter.

bool enableIntervalTrigger
Enables the DAC interval trigger.

2.34 PIT: Periodic Interrupt Timer

void PIT_Init(PIT_Type *base, const *pit_config_t* *config)
Ungates the PIT clock, enables the PIT module, and configures the peripheral for basic operations.

Note: This API should be called at the beginning of the application using the PIT driver.

Parameters

- base – PIT peripheral base address
- config – Pointer to the user's PIT config structure

void PIT_Deinit(PIT_Type *base)
Gates the PIT clock and disables the PIT module.

Parameters

- base – PIT peripheral base address

static inline void PIT_GetDefaultConfig(*pit_config_t* *config)
Fills in the PIT configuration structure with the default settings.
The default values are as follows.

```
config->enableRunInDebug = false;
```

Parameters

- config – Pointer to the configuration structure.

static inline void PIT_SetTimerChainMode(PIT_Type *base, *pit_chnl_t* channel, bool enable)

Enables or disables chaining a timer with the previous timer.

When a timer has a chain mode enabled, it only counts after the previous timer has expired. If the timer n-1 has counted down to 0, counter n decrements the value by one. Each timer is 32-bits, which allows the developers to chain timers together and form a longer timer (64-bits and larger). The first timer (timer 0) can't be chained to any other timer.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number which is chained with the previous timer
- enable – Enable or disable chain. true: Current timer is chained with the previous timer. false: Timer doesn't chain with other timers.

static inline void PIT_EnableInterrupts(PIT_Type *base, *pit_chnl_t* channel, uint32_t mask)

Enables the selected PIT interrupts.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number
- mask – The interrupts to enable. This is a logical OR of members of the enumeration `pit_interrupt_enable_t`

static inline void PIT_DisableInterrupts(PIT_Type *base, *pit_chnl_t* channel, uint32_t mask)

Disables the selected PIT interrupts.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number
- mask – The interrupts to disable. This is a logical OR of members of the enumeration `pit_interrupt_enable_t`

static inline uint32_t PIT_GetEnabledInterrupts(PIT_Type *base, *pit_chnl_t* channel)

Gets the enabled PIT interrupts.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `pit_interrupt_enable_t`

static inline uint32_t PIT_GetStatusFlags(PIT_Type *base, *pit_chnl_t* channel)

Gets the PIT status flags.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number

Returns

The status flags. This is the logical OR of members of the enumeration `pit_status_flags_t`

```
static inline void PIT_ClearStatusFlags(PIT_Type *base, pit_chnl_t channel, uint32_t mask)
```

Clears the PIT status flags.

Parameters

- *base* – PIT peripheral base address
- *channel* – Timer channel number
- *mask* – The status flags to clear. This is a logical OR of members of the enumeration `pit_status_flags_t`

```
static inline void PIT_SetTimerPeriod(PIT_Type *base, pit_chnl_t channel, uint32_t count)
```

Sets the timer period in units of count.

Timers begin counting from the value set by this function until it reaches 0, then it generates an interrupt and load this register value again. Writing a new value to this register does not restart the timer. Instead, the value is loaded after the timer expires.

Note: Users can call the utility macros provided in `fsl_common.h` to convert to ticks.

Parameters

- *base* – PIT peripheral base address
- *channel* – Timer channel number
- *count* – Timer period in units of ticks

```
static inline uint32_t PIT_GetCurrentTimerCount(PIT_Type *base, pit_chnl_t channel)
```

Reads the current timer counting value.

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note: Users can call the utility macros provided in `fsl_common.h` to convert ticks to usec or msec.

Parameters

- *base* – PIT peripheral base address
- *channel* – Timer channel number

Returns

Current timer counting value in ticks

```
static inline void PIT_StartTimer(PIT_Type *base, pit_chnl_t channel)
```

Starts the timer counting.

After calling this function, timers load period value, count down to 0 and then load the respective start value again. Each time a timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

Parameters

- *base* – PIT peripheral base address
- *channel* – Timer channel number.

```
static inline void PIT_StopTimer(PIT_Type *base, pit_chnl_t channel)
```

Stops the timer counting.

This function stops every timer counting. Timers reload their periods respectively after the next time they call the `PIT_DRV_StartTimer`.

Parameters

- base – PIT peripheral base address
- channel – Timer channel number.

FSL_PIT_DRIVER_VERSION

PIT Driver Version 2.2.0.

enum __pit_chnl

List of PIT channels.

Note: Actual number of available channels is SoC dependent

Values:

enumerator kPIT_Chnl_0

PIT channel number 0

enumerator kPIT_Chnl_1

PIT channel number 1

enumerator kPIT_Chnl_2

PIT channel number 2

enumerator kPIT_Chnl_3

PIT channel number 3

enum __pit_interrupt_enable

List of PIT interrupts.

Values:

enumerator kPIT_TimerInterruptEnable

Timer interrupt enable

enum __pit_status_flags

List of PIT status flags.

Values:

enumerator kPIT_TimerFlag

Timer flag

typedef enum __pit_chnl pit_chnl_t

List of PIT channels.

Note: Actual number of available channels is SoC dependent

typedef enum __pit_interrupt_enable pit_interrupt_enable_t

List of PIT interrupts.

typedef enum __pit_status_flags pit_status_flags_t

List of PIT status flags.

typedef struct __pit_config pit_config_t

PIT configuration structure.

This structure holds the configuration settings for the PIT peripheral. To initialize this structure to reasonable defaults, call the PIT_GetDefaultConfig() function and pass a pointer to your config structure instance.

The configuration structure can be made constant so it resides in flash.

uint64_t PIT_GetLifetimeTimerCount(PIT_Type *base)

Reads the current lifetime counter value.

The lifetime timer is a 64-bit timer which chains timer 0 and timer 1 together. Timer 0 and 1 are chained by calling the PIT_SetTimerChainMode before using this timer. The period of lifetime timer is equal to the “period of timer 0 * period of timer 1”. For the 64-bit value, the higher 32-bit has the value of timer 1, and the lower 32-bit has the value of timer 0.

Parameters

- base – PIT peripheral base address

Returns

Current lifetime timer value

struct __pit_config

#include <fsl_pit.h> PIT configuration structure.

This structure holds the configuration settings for the PIT peripheral. To initialize this structure to reasonable defaults, call the PIT_GetDefaultConfig() function and pass a pointer to your config structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

bool enableRunInDebug

true: Timers run in debug mode; false: Timers stop in debug mode

2.35 PMC: Power Management Controller

static inline void PMC_GetVersionId(PMC_Type *base, pmc_version_id_t *versionId)

Gets the PMC version ID.

This function gets the PMC version ID, including major version number, minor version number, and a feature specification number.

Parameters

- base – PMC peripheral base address.
- versionId – Pointer to version ID structure.

void PMC_GetParam(PMC_Type *base, pmc_param_t *param)

Gets the PMC parameter.

This function gets the PMC parameter including the VLPO enable and the HVD enable.

Parameters

- base – PMC peripheral base address.
- param – Pointer to PMC param structure.

void PMC_ConfigureLowVoltDetect(PMC_Type *base, const pmc_low_volt_detect_config_t *config)

Configures the low-voltage detect setting.

This function configures the low-voltage detect setting, including the trip point voltage setting, enables or disables the interrupt, enables or disables the system reset.

Parameters

- base – PMC peripheral base address.

- `config` – Low-voltage detect configuration structure.

`static inline bool PMC_GetLowVoltDetectFlag(PMC_Type *base)`

Gets the Low-voltage Detect Flag status.

This function reads the current LVDF status. If it returns 1, a low-voltage event is detected.

Parameters

- `base` – PMC peripheral base address.

Returns

Current low-voltage detect flag

- `true`: Low-voltage detected
- `false`: Low-voltage not detected

`static inline void PMC_ClearLowVoltDetectFlag(PMC_Type *base)`

Acknowledges clearing the Low-voltage Detect flag.

This function acknowledges the low-voltage detection errors (write 1 to clear LVDF).

Parameters

- `base` – PMC peripheral base address.

`void PMC_ConfigureLowVoltWarning(PMC_Type *base, const pmc_low_volt_warning_config_t *config)`

Configures the low-voltage warning setting.

This function configures the low-voltage warning setting, including the trip point voltage setting and enabling or disabling the interrupt.

Parameters

- `base` – PMC peripheral base address.
- `config` – Low-voltage warning configuration structure.

`static inline bool PMC_GetLowVoltWarningFlag(PMC_Type *base)`

Gets the Low-voltage Warning Flag status.

This function polls the current LVWF status. When 1 is returned, it indicates a low-voltage warning event. LVWF is set when V Supply transitions below the trip point or after reset and V Supply is already below the V LVW.

Parameters

- `base` – PMC peripheral base address.

Returns

Current LVWF status

- `true`: Low-voltage Warning Flag is set.
- `false`: the Low-voltage Warning does not happen.

`static inline void PMC_ClearLowVoltWarningFlag(PMC_Type *base)`

Acknowledges the Low-voltage Warning flag.

This function acknowledges the low voltage warning errors (write 1 to clear LVWF).

Parameters

- `base` – PMC peripheral base address.

```
void PMC_ConfigureHighVoltDetect(PMC_Type *base, const pmc_high_volt_detect_config_t
                                *config)
```

Configures the high-voltage detect setting.

This function configures the high-voltage detect setting, including the trip point voltage setting, enabling or disabling the interrupt, enabling or disabling the system reset.

Parameters

- base – PMC peripheral base address.
- config – High-voltage detect configuration structure.

```
static inline bool PMC_GetHighVoltDetectFlag(PMC_Type *base)
```

Gets the High-voltage Detect Flag status.

This function reads the current HVDF status. If it returns 1, a low voltage event is detected.

Parameters

- base – PMC peripheral base address.

Returns

Current high-voltage detect flag

- true: High-voltage detected
- false: High-voltage not detected

```
static inline void PMC_ClearHighVoltDetectFlag(PMC_Type *base)
```

Acknowledges clearing the High-voltage Detect flag.

This function acknowledges the high-voltage detection errors (write 1 to clear HVDF).

Parameters

- base – PMC peripheral base address.

```
void PMC_ConfigureBandgapBuffer(PMC_Type *base, const pmc_bandgap_buffer_config_t
                                *config)
```

Configures the PMC bandgap.

This function configures the PMC bandgap, including the drive select and behavior in low-power mode.

Parameters

- base – PMC peripheral base address.
- config – Pointer to the configuration structure

```
static inline bool PMC_GetPeriphIOIsolationFlag(PMC_Type *base)
```

Gets the acknowledge Peripherals and I/O pads isolation flag.

This function reads the Acknowledge Isolation setting that indicates whether certain peripherals and the I/O pads are in a latched state as a result of having been in the VLLS mode.

Parameters

- base – PMC peripheral base address.
- base – Base address for current PMC instance.

Returns

ACK isolation 0 - Peripherals and I/O pads are in a normal run state. 1 - Certain peripherals and I/O pads are in an isolated and latched state.

static inline void PMC_ClearPeriphIOIsolationFlag(PMC_Type *base)

Acknowledges the isolation flag to Peripherals and I/O pads.

This function clears the ACK Isolation flag. Writing one to this setting when it is set releases the I/O pads and certain peripherals to their normal run mode state.

Parameters

- base – PMC peripheral base address.

static inline bool PMC_IsRegulatorInRunRegulation(PMC_Type *base)

Gets the regulator regulation status.

This function returns the regulator to run a regulation status. It provides the current status of the internal voltage regulator.

Parameters

- base – PMC peripheral base address.
- base – Base address for current PMC instance.

Returns

Regulation status 0 - Regulator is in a stop regulation or in transition to/from the regulation. 1 - Regulator is in a run regulation.

FSL_PMC_DRIVER_VERSION

PMC driver version.

Version 2.0.3.

enum _pmc_low_volt_detect_volt_select

Low-voltage Detect Voltage Select.

Values:

enumerator kPMC_LowVoltDetectLowTrip

Low-trip point selected (VLVD = VLVDL)

enumerator kPMC_LowVoltDetectHighTrip

High-trip point selected (VLVD = VLVDH)

enum _pmc_low_volt_warning_volt_select

Low-voltage Warning Voltage Select.

Values:

enumerator kPMC_LowVoltWarningLowTrip

Low-trip point selected (VLVW = VLVW1)

enumerator kPMC_LowVoltWarningMid1Trip

Mid 1 trip point selected (VLVW = VLVW2)

enumerator kPMC_LowVoltWarningMid2Trip

Mid 2 trip point selected (VLVW = VLVW3)

enumerator kPMC_LowVoltWarningHighTrip

High-trip point selected (VLVW = VLVW4)

enum _pmc_high_volt_detect_volt_select

High-voltage Detect Voltage Select.

Values:

enumerator kPMC_HighVoltDetectLowTrip

Low-trip point selected (VHVD = VHVDL)

```
    enumerator kPMC_HighVoltDetectHighTrip
        High-trip point selected (VHVD = VHVDH )
enum __pmc_bandgap_buffer_drive_select
    Bandgap Buffer Drive Select.
    Values:
    enumerator kPMC_BandgapBufferDriveLow
        Low-drive.
    enumerator kPMC_BandgapBufferDriveHigh
        High-drive.
enum __pmc_vlp_freq_option
    VLPx Option.
    Values:
    enumerator kPMC_FreqRestrict
        Frequency is restricted in VLPx mode.
    enumerator kPMC_FreqUnrestrict
        Frequency is unrestricted in VLPx mode.
typedef enum __pmc_low_volt_detect_volt_select pmc_low_volt_detect_volt_select_t
    Low-voltage Detect Voltage Select.
typedef enum __pmc_low_volt_warning_volt_select pmc_low_volt_warning_volt_select_t
    Low-voltage Warning Voltage Select.
typedef enum __pmc_high_volt_detect_volt_select pmc_high_volt_detect_volt_select_t
    High-voltage Detect Voltage Select.
typedef enum __pmc_bandgap_buffer_drive_select pmc_bandgap_buffer_drive_select_t
    Bandgap Buffer Drive Select.
typedef enum __pmc_vlp_freq_option pmc_vlp_freq_mode_t
    VLPx Option.
typedef struct __pmc_version_id pmc_version_id_t
    IP version ID definition.
typedef struct __pmc_param pmc_param_t
    IP parameter definition.
typedef struct __pmc_low_volt_detect_config pmc_low_volt_detect_config_t
    Low-voltage Detect Configuration Structure.
typedef struct __pmc_low_volt_warning_config pmc_low_volt_warning_config_t
    Low-voltage Warning Configuration Structure.
typedef struct __pmc_high_volt_detect_config pmc_high_volt_detect_config_t
    High-voltage Detect Configuration Structure.
typedef struct __pmc_bandgap_buffer_config pmc_bandgap_buffer_config_t
    Bandgap Buffer configuration.
struct __pmc_version_id
    #include <fsl_pmc.h> IP version ID definition.
```

Public Members

uint16_t feature
Feature Specification Number.

uint8_t minor
Minor version number.

uint8_t major
Major version number.

struct __pmc_param
#include <fsl_pmc.h> IP parameter definition.

Public Members

bool vlpoEnable
VLPO enable.

bool hvdEnable
HVD enable.

struct __pmc_low_volt_detect_config
#include <fsl_pmc.h> Low-voltage Detect Configuration Structure.

Public Members

bool enableInt
Enable interrupt when Low-voltage detect

bool enableReset
Enable system reset when Low-voltage detect

pmc_low_volt_detect_volt_select_t voltSelect
Low-voltage detect trip point voltage selection

struct __pmc_low_volt_warning_config
#include <fsl_pmc.h> Low-voltage Warning Configuration Structure.

Public Members

bool enableInt
Enable interrupt when low-voltage warning

pmc_low_volt_warning_volt_select_t voltSelect
Low-voltage warning trip point voltage selection

struct __pmc_high_volt_detect_config
#include <fsl_pmc.h> High-voltage Detect Configuration Structure.

Public Members

bool enableInt
Enable interrupt when high-voltage detect

bool enableReset
Enable system reset when high-voltage detect

pmc_high_volt_detect_volt_select_t voltSelect
High-voltage detect trip point voltage selection

struct __pmc_bandgap_buffer_config
#include <fsl_pmc.h> Bandgap Buffer configuration.

Public Members

bool enable
Enable bandgap buffer.

bool enableInLowPowerMode
Enable bandgap buffer in low-power mode.

pmc_bandgap_buffer_drive_select_t drive
Bandgap buffer drive select.

2.36 PORT: Port Control and Interrupts

static inline void PORT_SetPinConfig(PORT_Type *base, uint32_t pin, const *port_pin_config_t* *config)

Sets the port PCR register.

This is an example to define an input pin or output pin PCR configuration.

```
// Define a digital input pin PCR configuration
port_pin_config_t config = {
    kPORT_PullUp,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainDisable,
    kPORT_LowDriveStrength,
    kPORT_MuxAsGpio,
    kPORT_UnLockRegister,
};
```

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- config – PORT PCR register configuration structure.

static inline void PORT_SetMultiplePinsConfig(PORT_Type *base, uint32_t mask, const *port_pin_config_t* *config)

Sets the port PCR register for multiple pins.

This is an example to define input pins or output pins PCR configuration.

```
Define a digital input pin PCR configuration
port_pin_config_t config = {
    kPORT_PullUp ,
    kPORT_PullEnable,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainDisable,
    kPORT_LowDriveStrength,
    kPORT_MuxAsGpio,
```

(continues on next page)

(continued from previous page)

```
kPORT_UnlockRegister,
};
```

Parameters

- base – PORT peripheral base pointer.
- mask – PORT pin number macro.
- config – PORT PCR register configuration structure.

```
static inline void PORT_SetMultipleInterruptPinsConfig(PORT_Type *base, uint32_t mask,
                                                    port_interrupt_t config)
```

Sets the port interrupt configuration in PCR register for multiple pins.

Parameters

- base – PORT peripheral base pointer.
- mask – PORT pin number macro.
- config – PORT pin interrupt configuration.
 - kPORT_InterruptOrDMADisabled: Interrupt/DMA request disabled.
 - kPORT_DMARisingEdge : DMA request on rising edge(if the DMA requests exit).
 - kPORT_DMAFallingEdge: DMA request on falling edge(if the DMA requests exit).
 - kPORT_DMAEitherEdge : DMA request on either edge(if the DMA requests exit).
 - kPORT_FlagRisingEdge : Flag sets on rising edge(if the Flag states exit).
 - kPORT_FlagFallingEdge : Flag sets on falling edge(if the Flag states exit).
 - kPORT_FlagEitherEdge : Flag sets on either edge(if the Flag states exit).
 - kPORT_InterruptLogicZero : Interrupt when logic zero.
 - kPORT_InterruptRisingEdge : Interrupt on rising edge.
 - kPORT_InterruptFallingEdge: Interrupt on falling edge.
 - kPORT_InterruptEitherEdge : Interrupt on either edge.
 - kPORT_InterruptLogicOne : Interrupt when logic one.
 - kPORT_ActiveHighTriggerOutputEnable : Enable active high-trigger output (if the trigger states exit).
 - kPORT_ActiveLowTriggerOutputEnable : Enable active low-trigger output (if the trigger states exit).

```
static inline void PORT_SetPinMux(PORT_Type *base, uint32_t pin, port_mux_t mux)
```

Configures the pin muxing.

Note: : This function is NOT recommended to use together with the PORT_SetPinsConfig, because the PORT_SetPinsConfig need to configure the pin mux anyway (Otherwise the pin mux is reset to zero : kPORT_PinDisabledOrAnalog). This function is recommended to use to reset the pin mux

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- mux – pin muxing slot selection.
 - kPORT_PinDisabledOrAnalog: Pin disabled or work in analog function.
 - kPORT_MuxAsGpio : Set as GPIO.
 - kPORT_MuxAlt2 : chip-specific.
 - kPORT_MuxAlt3 : chip-specific.
 - kPORT_MuxAlt4 : chip-specific.
 - kPORT_MuxAlt5 : chip-specific.
 - kPORT_MuxAlt6 : chip-specific.
 - kPORT_MuxAlt7 : chip-specific.

static inline void PORT__EnablePinsDigitalFilter(PORT_Type *base, uint32_t mask, bool enable)
Enables the digital filter in one port, each bit of the 32-bit register represents one pin.

Parameters

- base – PORT peripheral base pointer.
- mask – PORT pin number macro.
- enable – PORT digital filter configuration.

static inline void PORT__SetDigitalFilterConfig(PORT_Type *base, const
port_digital_filter_config_t *config)

Sets the digital filter in one port, each bit of the 32-bit register represents one pin.

Parameters

- base – PORT peripheral base pointer.
- config – PORT digital filter configuration structure.

static inline void PORT__SetPinInterruptConfig(PORT_Type *base, uint32_t pin, *port_interrupt_t*
config)

Configures the port pin interrupt/DMA request.

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- config – PORT pin interrupt configuration.
 - kPORT_InterruptOrDMADisabled: Interrupt/DMA request disabled.
 - kPORT_DMARisingEdge : DMA request on rising edge(if the DMA requests exit).
 - kPORT_DMAFallingEdge: DMA request on falling edge(if the DMA requests exit).
 - kPORT_DMAEitherEdge : DMA request on either edge(if the DMA requests exit).
 - kPORT_FlagRisingEdge : Flag sets on rising edge(if the Flag states exit).
 - kPORT_FlagFallingEdge : Flag sets on falling edge(if the Flag states exit).
 - kPORT_FlagEitherEdge : Flag sets on either edge(if the Flag states exit).
 - kPORT_InterruptLogicZero : Interrupt when logic zero.

- kPORT_InterruptRisingEdge : Interrupt on rising edge.
- kPORT_InterruptFallingEdge: Interrupt on falling edge.
- kPORT_InterruptEitherEdge : Interrupt on either edge.
- kPORT_InterruptLogicOne : Interrupt when logic one.
- kPORT_ActiveHighTriggerOutputEnable : Enable active high-trigger output (if the trigger states exit).
- kPORT_ActiveLowTriggerOutputEnable : Enable active low-trigger output (if the trigger states exit).

static inline void PORT_SetPinDriveStrength(PORT_Type *base, uint32_t pin, uint8_t strength)
Configures the port pin drive strength.

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- strength – PORT pin drive strength
 - kPORT_LowDriveStrength = 0U - Low-drive strength is configured.
 - kPORT_HighDriveStrength = 1U - High-drive strength is configured.

static inline uint32_t PORT_GetPinsInterruptFlags(PORT_Type *base)

Reads the whole port status flag.

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

- base – PORT peripheral base pointer.

Returns

Current port interrupt status flags, for example, 0x00010001 means the pin 0 and 16 have the interrupt.

static inline void PORT_ClearPinsInterruptFlags(PORT_Type *base, uint32_t mask)

Clears the multiple pin interrupt status flag.

Parameters

- base – PORT peripheral base pointer.
- mask – PORT pin number macro.

FSL_PORT_DRIVER_VERSION

PORT driver version.

enum __port_pull

Internal resistor pull feature selection.

Values:

enumerator kPORT_PullDisable

Internal pull-up/down resistor is disabled.

enumerator kPORT_PullDown

Internal pull-down resistor is enabled.

enumerator kPORT_PullUp
Internal pull-up resistor is enabled.

enum __port_slew_rate
Slew rate selection.

Values:

enumerator kPORT_FastSlewRate
Fast slew rate is configured.

enumerator kPORT_SlowSlewRate
Slow slew rate is configured.

enum __port_open_drain_enable
Open Drain feature enable/disable.

Values:

enumerator kPORT_OpenDrainDisable
Open drain output is disabled.

enumerator kPORT_OpenDrainEnable
Open drain output is enabled.

enum __port_passive_filter_enable
Passive filter feature enable/disable.

Values:

enumerator kPORT_PassiveFilterDisable
Passive input filter is disabled.

enumerator kPORT_PassiveFilterEnable
Passive input filter is enabled.

enum __port_drive_strength
Configures the drive strength.

Values:

enumerator kPORT_LowDriveStrength
Low-drive strength is configured.

enumerator kPORT_HighDriveStrength
High-drive strength is configured.

enum __port_lock_register
Unlock/lock the pin control register field[15:0].

Values:

enumerator kPORT_UnlockRegister
Pin Control Register fields [15:0] are not locked.

enumerator kPORT_LockRegister
Pin Control Register fields [15:0] are locked.

enum __port_mux
Pin mux selection.

Values:

enumerator kPORT_PinDisabledOrAnalog
Corresponding pin is disabled, but is used as an analog pin.

enumerator kPORT_MuxAsGpio
Corresponding pin is configured as GPIO.

enumerator kPORT_MuxAlt0
Chip-specific

enumerator kPORT_MuxAlt1
Chip-specific

enumerator kPORT_MuxAlt2
Chip-specific

enumerator kPORT_MuxAlt3
Chip-specific

enumerator kPORT_MuxAlt4
Chip-specific

enumerator kPORT_MuxAlt5
Chip-specific

enumerator kPORT_MuxAlt6
Chip-specific

enumerator kPORT_MuxAlt7
Chip-specific

enumerator kPORT_MuxAlt8
Chip-specific

enumerator kPORT_MuxAlt9
Chip-specific

enumerator kPORT_MuxAlt10
Chip-specific

enumerator kPORT_MuxAlt11
Chip-specific

enumerator kPORT_MuxAlt12
Chip-specific

enumerator kPORT_MuxAlt13
Chip-specific

enumerator kPORT_MuxAlt14
Chip-specific

enumerator kPORT_MuxAlt15
Chip-specific

enum __port_interrupt
Configures the interrupt generation condition.

Values:

enumerator kPORT_InterruptOrDMADisabled
Interrupt/DMA request is disabled.

enumerator kPORT_DMARisingEdge
DMA request on rising edge.

enumerator kPORT_DMAFallingEdge
DMA request on falling edge.

enumerator kPORT_DMAEitherEdge
DMA request on either edge.

enumerator kPORT_FlagRisingEdge
Flag sets on rising edge.

enumerator kPORT_FlagFallingEdge
Flag sets on falling edge.

enumerator kPORT_FlagEitherEdge
Flag sets on either edge.

enumerator kPORT_InterruptLogicZero
Interrupt when logic zero.

enumerator kPORT_InterruptRisingEdge
Interrupt on rising edge.

enumerator kPORT_InterruptFallingEdge
Interrupt on falling edge.

enumerator kPORT_InterruptEitherEdge
Interrupt on either edge.

enumerator kPORT_InterruptLogicOne
Interrupt when logic one.

enumerator kPORT_ActiveHighTriggerOutputEnable
Enable active high-trigger output.

enumerator kPORT_ActiveLowTriggerOutputEnable
Enable active low-trigger output.

enum _port_digital_filter_clock_source
Digital filter clock source selection.

Values:

enumerator kPORT_BusClock
Digital filters are clocked by the bus clock.

enumerator kPORT_LpoClock
Digital filters are clocked by the 1 kHz LPO clock.

typedef enum _port_mux port_mux_t
Pin mux selection.

typedef enum _port_interrupt port_interrupt_t
Configures the interrupt generation condition.

typedef enum _port_digital_filter_clock_source port_digital_filter_clock_source_t
Digital filter clock source selection.

typedef struct _port_digital_filter_config port_digital_filter_config_t
PORT digital filter feature configuration definition.

typedef struct _port_pin_config port_pin_config_t
PORT pin configuration structure.

FSL_COMPONENT_ID

```
struct __port_digital_filter_config
#include <fsl_port.h> PORT digital filter feature configuration definition.
```

Public Members

```
uint32_t digitalFilterWidth
    Set digital filter width
port_digital_filter_clock_source_t clockSource
    Set digital filter clockSource
```

```
struct __port_pin_config
#include <fsl_port.h> PORT pin configuration structure.
```

Public Members

```
uint16_t pullSelect
    No-pull/pull-down/pull-up select
uint16_t slewRate
    Fast/slow slew rate Configure
uint16_t passiveFilterEnable
    Passive filter enable/disable
uint16_t openDrainEnable
    Open drain enable/disable
uint16_t driveStrength
    Fast/slow drive strength configure
uint16_t lockRegister
    Lock/unlock the PCR field[15:0]
```

2.37 QTMR: Quad Timer Driver

```
void QTMR_Init(TMR_Type *base, const qtmr_config_t *config)
    Ungates the Quad Timer clock and configures the peripheral for basic operation.
```

Note: This API should be called at the beginning of the application using the Quad Timer driver.

Parameters

- base – Quad Timer peripheral base address
- config – Pointer to user's Quad Timer config structure

```
void QTMR_Deinit(TMR_Type *base)
    Stops the counter and gates the Quad Timer clock.
```

Parameters

- base – Quad Timer peripheral base address

void QTMR_GetDefaultConfig(*qtmr_config_t* *config)

Fill in the Quad Timer config struct with the default settings.

The default values are:

```
config->debugMode = kQTMR_RunNormalInDebug;
config->enableExternalForce = false;
config->enableMasterMode = false;
config->faultFilterCount = 0;
config->faultFilterPeriod = 0;
config->primarySource = kQTMR_ClockDivide_2;
config->secondarySource = kQTMR_Counter0InputPin;
```

Parameters

- config – Pointer to user's Quad Timer config structure.

void QTMR_EnableInterrupts(TMR_Type *base, uint32_t mask)

Enables the selected Quad Timer interrupts.

Parameters

- base – Quad Timer peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration `qtmr_interrupt_enable_t`

void QTMR_DisableInterrupts(TMR_Type *base, uint32_t mask)

Disables the selected Quad Timer interrupts.

Parameters

- base – Quad Timer peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration `qtmr_interrupt_enable_t`

uint32_t QTMR_GetEnabledInterrupts(TMR_Type *base)

Gets the enabled Quad Timer interrupts.

Parameters

- base – Quad Timer peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `qtmr_interrupt_enable_t`

uint32_t QTMR_GetStatus(TMR_Type *base)

Gets the Quad Timer status flags.

Parameters

- base – Quad Timer peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration `qtmr_status_flags_t`

void QTMR_ClearStatusFlags(TMR_Type *base, uint32_t mask)

Clears the Quad Timer status flags.

Parameters

- base – Quad Timer peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration `qtmr_status_flags_t`

`void QTMR_SetTimerPeriod(TMR_Type *base, uint16_t ticks)`

Sets the timer period in ticks.

Timers counts from initial value till it equals the count value set here. The counter will then reinitialize to the value specified in the Load register.

Note:

- a. This function will write the time period in ticks to COMP1 or COMP2 register depending on the count direction
 - b. User can call the utility macros provided in `fsl_common.h` to convert to ticks
 - c. This function supports cases, providing only primary source clock without secondary source clock.
-

Parameters

- `base` – Quad Timer peripheral base address
- `ticks` – Timer period in units of ticks

`static inline uint16_t QTMR_GetCurrentTimerCount(TMR_Type *base)`

Reads the current timer counting value.

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note: User can call the utility macros provided in `fsl_common.h` to convert ticks to usec or msec

Parameters

- `base` – Quad Timer peripheral base address

Returns

Current counter value in ticks

`static inline void QTMR_StartTimer(TMR_Type *base, qtmr_counting_mode_t clockSource)`

Starts the Quad Timer counter.

Parameters

- `base` – Quad Timer peripheral base address
- `clockSource` – Quad Timer clock source

`static inline void QTMR_StopTimer(TMR_Type *base)`

Stops the Quad Timer counter.

Parameters

- `base` – Quad Timer peripheral base address

`FSL_QTMR_DRIVER_VERSION`

Version.

`enum __qtmr_primary_count_source`

Quad Timer primary clock source selection.

Values:

enumerator `kQTMR_ClockCounter0InputPin`

Use counter 0 input pin

enumerator kQTMR__ClockCounter1InputPin

Use counter 1 input pin

enumerator kQTMR__ClockCounter2InputPin

Use counter 2 input pin

enumerator kQTMR__ClockCounter3InputPin

Use counter 3 input pin

enumerator kQTMR__ClockCounter0Output

Use counter 0 output

enumerator kQTMR__ClockCounter1Output

Use counter 1 output

enumerator kQTMR__ClockCounter2Output

Use counter 2 output

enumerator kQTMR__ClockCounter3Output

Use counter 3 output

enumerator kQTMR__ClockDivide__1

IP bus clock divide by 1 prescaler

enumerator kQTMR__ClockDivide__2

IP bus clock divide by 2 prescaler

enumerator kQTMR__ClockDivide__4

IP bus clock divide by 4 prescaler

enumerator kQTMR__ClockDivide__8

IP bus clock divide by 8 prescaler

enumerator kQTMR__ClockDivide__16

IP bus clock divide by 16 prescaler

enumerator kQTMR__ClockDivide__32

IP bus clock divide by 32 prescaler

enumerator kQTMR__ClockDivide__64

IP bus clock divide by 64 prescaler

enumerator kQTMR__ClockDivide__128

IP bus clock divide by 128 prescaler

enum __qtmr_input_source

Quad Timer input sources selection.

Values:

enumerator kQTMR__Counter0InputPin

Use counter 0 input pin

enumerator kQTMR__Counter1InputPin

Use counter 1 input pin

enumerator kQTMR__Counter2InputPin

Use counter 2 input pin

enumerator kQTMR__Counter3InputPin

Use counter 3 input pin

enum _qtmr_counting_mode

Quad Timer counting mode selection.

Values:

enumerator kQTMR_NoOperation

No operation

enumerator kQTMR_PriSrcRiseEdge

Count rising edges of primary source

enumerator kQTMR_PriSrcRiseAndFallEdge

Count rising and falling edges of primary source

enumerator kQTMR_PriSrcRiseEdgeSecInpHigh

Count rise edges of pri SRC while sec inp high active

enumerator kQTMR_QuadCountMode

Quadrature count mode, uses pri and sec sources

enumerator kQTMR_PriSrcRiseEdgeSecDir

Count rising edges of pri SRC; sec SRC specifies dir

enumerator kQTMR_SecSrcTrigPriCnt

Edge of sec SRC trigger primary count until compare

enumerator kQTMR_CascadeCount

Cascaded count mode (up/down)

enum _qtmr_output_mode

Quad Timer output mode selection.

Values:

enumerator kQTMR_AssertWhenCountActive

Assert OFLAG while counter is active

enumerator kQTMR_ClearOnCompare

Clear OFLAG on successful compare

enumerator kQTMR_SetOnCompare

Set OFLAG on successful compare

enumerator kQTMR_ToggleOnCompare

Toggle OFLAG on successful compare

enumerator kQTMR_ToggleOnAltCompareReg

Toggle OFLAG using alternating compare registers

enumerator kQTMR_SetOnCompareClearOnSecSrcInp

Set OFLAG on compare, clear on sec SRC input edge

enumerator kQTMR_SetOnCompareClearOnCountRoll

Set OFLAG on compare, clear on counter rollover

enumerator kQTMR_EnableGateClock

Enable gated clock output while count is active

enum _qtmr_input_capture_edge

Quad Timer input capture edge mode, rising edge, or falling edge.

Values:

enumerator kQTMR_NoCapture

Capture is disabled

enumerator kQTMR_RisingEdge

Capture on rising edge (IPS=0) or falling edge (IPS=1)

enumerator kQTMR_FallingEdge

Capture on falling edge (IPS=0) or rising edge (IPS=1)

enumerator kQTMR_RisingAndFallingEdge

Capture on both edges

enum __qtmr_preload_control

Quad Timer input capture edge mode, rising edge, or falling edge.

Values:

enumerator kQTMR_NoPreload

Never preload

enumerator kQTMR_LoadOnComp1

Load upon successful compare with value in COMP1

enumerator kQTMR_LoadOnComp2

Load upon successful compare with value in COMP2

enum __qtmr_debug_action

List of Quad Timer run options when in Debug mode.

Values:

enumerator kQTMR_RunNormalInDebug

Continue with normal operation

enumerator kQTMR_HaltCounter

Halt counter

enumerator kQTMR_ForceOutToZero

Force output to logic 0

enumerator kQTMR_HaltCountForceOutZero

Halt counter and force output to logic 0

enum __qtmr_interrupt_enable

List of Quad Timer interrupts.

Values:

enumerator kQTMR_CompareInterruptEnable

Compare interrupt.

enumerator kQTMR_Compare1InterruptEnable

Compare 1 interrupt.

enumerator kQTMR_Compare2InterruptEnable

Compare 2 interrupt.

enumerator kQTMR_OverflowInterruptEnable

Timer overflow interrupt.

enumerator kQTMR_EdgeInterruptEnable

Input edge interrupt.

enum `_qtmr_status_flags`

List of Quad Timer flags.

Values:

enumerator `kQTMR_CompareFlag`

Compare flag

enumerator `kQTMR_Compare1Flag`

Compare 1 flag

enumerator `kQTMR_Compare2Flag`

Compare 2 flag

enumerator `kQTMR_OverflowFlag`

Timer overflow flag

enumerator `kQTMR_EdgeFlag`

Input edge flag

typedef enum `_qtmr_primary_count_source` `qtmr_primary_count_source_t`

Quad Timer primary clock source selection.

typedef enum `_qtmr_input_source` `qtmr_input_source_t`

Quad Timer input sources selection.

typedef enum `_qtmr_counting_mode` `qtmr_counting_mode_t`

Quad Timer counting mode selection.

typedef enum `_qtmr_output_mode` `qtmr_output_mode_t`

Quad Timer output mode selection.

typedef enum `_qtmr_input_capture_edge` `qtmr_input_capture_edge_t`

Quad Timer input capture edge mode, rising edge, or falling edge.

typedef enum `_qtmr_preload_control` `qtmr_preload_control_t`

Quad Timer input capture edge mode, rising edge, or falling edge.

typedef enum `_qtmr_debug_action` `qtmr_debug_action_t`

List of Quad Timer run options when in Debug mode.

typedef enum `_qtmr_interrupt_enable` `qtmr_interrupt_enable_t`

List of Quad Timer interrupts.

typedef enum `_qtmr_status_flags` `qtmr_status_flags_t`

List of Quad Timer flags.

typedef struct `_qtmr_config` `qtmr_config_t`

Quad Timer config structure.

This structure holds the configuration settings for the Quad Timer peripheral. To initialize this structure to reasonable defaults, call the `QTMR_GetDefaultConfig()` function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

`status_t` `QTMR_SetupPwm(TMR_Type *base, uint32_t pwmFreqHz, uint8_t dutyCyclePercent, bool outputPolarity, uint32_t srcClock_Hz)`

Sets up Quad timer module for PWM signal output.

The function initializes the timer module according to the parameters passed in by the user. The function also sets up the value compare registers to match the PWM signal requirements.

Parameters

- base – Quad Timer peripheral base address
- pwmFreqHz – PWM signal frequency in Hz
- dutyCyclePercent – PWM pulse width, value should be between 0 to 100
0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)
- outputPolarity – true: invert polarity of the output signal, false: no inversion
- srcClock_Hz – Main counter clock in Hz.

Returns

Returns an error if there was error setting up the signal.

```
void QTMR_SetupInputCapture(TMR_Type *base, qtmr_input_source_t capturePin, bool  
                           inputPolarity, bool reloadOnCapture,  
                           qtmr_input_capture_edge_t captureMode)
```

Allows the user to count the source clock cycles until a capture event arrives.

The count is stored in the capture register.

Parameters

- base – Quad Timer peripheral base address
- capturePin – Pin through which we receive the input signal to trigger the capture
- inputPolarity – true: invert polarity of the input signal, false: no inversion
- reloadOnCapture – true: reload the counter when an input capture occurs, false: no reload
- captureMode – Specifies which edge of the input signal triggers a capture

```
struct _qtmr_config
```

#include <fsl_qtmr.h> Quad Timer config structure.

This structure holds the configuration settings for the Quad Timer peripheral. To initialize this structure to reasonable defaults, call the QTMR_GetDefaultConfig() function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

Public Members

```
qtmr_primary_count_source_t primarySource
```

Specify the primary count source

```
qtmr_input_source_t secondarySource
```

Specify the secondary count source

```
bool enableMasterMode
```

true: Broadcast compare function output to other counters; false no broadcast

```
bool enableExternalForce
```

true: Compare from another counter force state of OFLAG signal false: OFLAG controlled by local counter

```
uint8_t faultFilterCount
```

Fault filter count

```
uint8_t faultFilterPeriod
```

Fault filter period;value of 0 will bypass the filter

qtmr_debug_action_t debugMode
Operation in Debug mode

2.38 RCM: Reset Control Module Driver

static inline void RCM_GetVersionId(RCM_Type *base, *rcm_version_id_t* *versionId)

Gets the RCM version ID.

This function gets the RCM version ID including the major version number, the minor version number, and the feature specification number.

Parameters

- base – RCM peripheral base address.
- versionId – Pointer to the version ID structure.

static inline uint32_t RCM_GetResetSourceImplementedStatus(RCM_Type *base)

Gets the reset source implemented status.

This function gets the RCM parameter that indicates whether the corresponding reset source is implemented. Use source masks defined in the *rcm_reset_source_t* to get the desired source status.

This is an example.

```
uint32_t status;
```

To test whether the MCU is reset using Watchdog.

```
status = RCM_GetResetSourceImplementedStatus(RCM) & (kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

- base – RCM peripheral base address.

Returns

All reset source implemented status bit map.

static inline uint32_t RCM_GetPreviousResetSources(RCM_Type *base)

Gets the reset source status which caused a previous reset.

This function gets the current reset source status. Use source masks defined in the *rcm_reset_source_t* to get the desired source status.

This is an example.

```
uint32_t resetStatus;
```

To get all reset source statuses.

```
resetStatus = RCM_GetPreviousResetSources(RCM) & kRCM_SourceAll;
```

To test whether the MCU is reset using Watchdog.

```
resetStatus = RCM_GetPreviousResetSources(RCM) & kRCM_SourceWdog;
```

To test multiple reset sources.

```
resetStatus = RCM_GetPreviousResetSources(RCM) & (kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

- base – RCM peripheral base address.

Returns

All reset source status bit map.

```
static inline uint32_t RCM_GetStickyResetSources(RCM_Type *base)
```

Gets the sticky reset source status.

This function gets the current reset source status that has not been cleared by software for a specific source.

This is an example.

```
uint32_t resetStatus;
```

To get **all** reset source statuses.

```
resetStatus = RCM_GetStickyResetSources(RCM) & kRCM_SourceAll;
```

To test whether the MCU **is** reset using Watchdog.

```
resetStatus = RCM_GetStickyResetSources(RCM) & kRCM_SourceWdog;
```

To test multiple reset sources.

```
resetStatus = RCM_GetStickyResetSources(RCM) & (kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

- base – RCM peripheral base address.

Returns

All reset source status bit map.

```
static inline void RCM_ClearStickyResetSources(RCM_Type *base, uint32_t sourceMasks)
```

Clears the sticky reset source status.

This function clears the sticky system reset flags indicated by source masks.

This is an example.

Clears multiple reset sources.

```
RCM_ClearStickyResetSources(kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

- base – RCM peripheral base address.
- sourceMasks – reset source status bit map

```
void RCM_ConfigureResetPinFilter(RCM_Type *base, const rcm_reset_pin_filter_config_t *config)
```

Configures the reset pin filter.

This function sets the reset pin filter including the filter source, filter width, and so on.

Parameters

- base – RCM peripheral base address.
- config – Pointer to the configuration structure.

```
static inline bool RCM_GetEasyPortModePinStatus(RCM_Type *base)
```

Gets the EZP_MS_B pin assert status.

This function gets the easy port mode status (EZP_MS_B) pin assert status.

Parameters

- base – RCM peripheral base address.

Returns

status true - asserted, false - reasserted

```
static inline rcm_boot_rom_config_t RCM_GetBootRomSource(RCM_Type *base)
```

Gets the ROM boot source.

This function gets the ROM boot source during the last chip reset.

Parameters

- *base* – RCM peripheral base address.

Returns

The ROM boot source.

```
static inline void RCM_ClearBootRomSource(RCM_Type *base)
```

Clears the ROM boot source flag.

This function clears the ROM boot source flag.

Parameters

- *base* – Register base address of RCM

```
void RCM_SetForceBootRomSource(RCM_Type *base, rcm_boot_rom_config_t config)
```

Forces the boot from ROM.

This function forces booting from ROM during all subsequent system resets.

Parameters

- *base* – RCM peripheral base address.
- *config* – Boot configuration.

```
static inline void RCM_SetSystemResetInterruptConfig(RCM_Type *base, uint32_t intMask,  
                                                    rcm_reset_delay_t delay)
```

Sets the system reset interrupt configuration.

For a graceful shut down, the RCM supports delaying the assertion of the system reset for a period of time when the reset interrupt is generated. This function can be used to enable the interrupt and the delay period. The interrupts are passed in as bit mask. See *rcm_int_t* for details. For example, to delay a reset for 512 LPO cycles after the WDOG timeout or loss-of-clock occurs, configure as follows: `RCM_SetSystemResetInterruptConfig(kRCM_IntWatchDog | kRCM_IntLossOfClk, kRCM_ResetDelay512Lpo);`

Parameters

- *base* – RCM peripheral base address.
- *intMask* – Bit mask of the system reset interrupts to enable. See *rcm_interrupt_enable_t* for details.
- *delay* – Bit mask of the system reset interrupts to enable.

```
FSL_RCM_DRIVER_VERSION
```

RCM driver version 2.0.4.

```
enum _rcm_reset_source
```

System Reset Source Name definitions.

Values:

```
enumerator kRCM_SourceWakeup
```

Low-leakage wakeup reset

```
enumerator kRCM_SourceLvd
```

Low-voltage detect reset

enumerator kRCM_SourceLoc
Loss of clock reset

enumerator kRCM_SourceLol
Loss of lock reset

enumerator kRCM_SourceWdog
Watchdog reset

enumerator kRCM_SourcePin
External pin reset

enumerator kRCM_SourcePor
Power on reset

enumerator kRCM_SourceJtag
JTAG generated reset

enumerator kRCM_SourceLockup
Core lock up reset

enumerator kRCM_SourceSw
Software reset

enumerator kRCM_SourceMdma
MDM-AP system reset

enumerator kRCM_SourceEzpt
EzPort reset

enumerator kRCM_SourceSackerr
Parameter could get all reset flags

enumerator kRCM_SourceAll

enum _rcm_run_wait_filter_mode
Reset pin filter select in Run and Wait modes.

Values:

enumerator kRCM_FilterDisable
All filtering disabled

enumerator kRCM_FilterBusClock
Bus clock filter enabled

enumerator kRCM_FilterLpoClock
LPO clock filter enabled

enum _rcm_boot_rom_config
Boot from ROM configuration.

Values:

enumerator kRCM_BootFlash
Boot from flash

enumerator kRCM_BootRomCfg0
Boot from boot ROM due to BOOTCFG0

enumerator kRCM_BootRomFopt
Boot from boot ROM due to FOPT[7]

enumerator kRCM_BootRomBoth

Boot from boot ROM due to both BOOTCFG0 and FOPT[7]

enum _rcm_reset_delay

Maximum delay time from interrupt asserts to system reset.

Values:

enumerator kRCM_ResetDelay8Lpo

Delay 8 LPO cycles.

enumerator kRCM_ResetDelay32Lpo

Delay 32 LPO cycles.

enumerator kRCM_ResetDelay128Lpo

Delay 128 LPO cycles.

enumerator kRCM_ResetDelay512Lpo

Delay 512 LPO cycles.

enum _rcm_interrupt_enable

System reset interrupt enable bit definitions.

Values:

enumerator kRCM_IntNone

No interrupt enabled.

enumerator kRCM_IntLossOfClk

Loss of clock interrupt.

enumerator kRCM_IntLossOfLock

Loss of lock interrupt.

enumerator kRCM_IntWatchDog

Watch dog interrupt.

enumerator kRCM_IntExternalPin

External pin interrupt.

enumerator kRCM_IntGlobal

Global interrupts.

enumerator kRCM_IntCoreLockup

Core lock up interrupt

enumerator kRCM_IntSoftware

software interrupt

enumerator kRCM_IntStopModeAckErr

Stop mode ACK error interrupt.

enumerator kRCM_IntCore1

Core 1 interrupt.

enumerator kRCM_IntAll

Enable all interrupts.

typedef enum _rcm_reset_source rcm_reset_source_t

System Reset Source Name definitions.

typedef enum _rcm_run_wait_filter_mode rcm_run_wait_filter_mode_t

Reset pin filter select in Run and Wait modes.

typedef enum *_rcm_boot_rom_config* rcm_boot_rom_config_t

Boot from ROM configuration.

typedef enum *_rcm_reset_delay* rcm_reset_delay_t

Maximum delay time from interrupt asserts to system reset.

typedef enum *_rcm_interrupt_enable* rcm_interrupt_enable_t

System reset interrupt enable bit definitions.

typedef struct *_rcm_version_id* rcm_version_id_t

IP version ID definition.

typedef struct *_rcm_reset_pin_filter_config* rcm_reset_pin_filter_config_t

Reset pin filter configuration.

struct *_rcm_version_id*

#include <fsl_rcm.h> IP version ID definition.

Public Members

uint16_t feature

Feature Specification Number.

uint8_t minor

Minor version number.

uint8_t major

Major version number.

struct *_rcm_reset_pin_filter_config*

#include <fsl_rcm.h> Reset pin filter configuration.

Public Members

bool enableFilterInStop

Reset pin filter select in stop mode.

rcm_run_wait_filter_mode_t filterInRunWait

Reset pin filter in run/wait mode.

uint8_t busClockFilterCount

Reset pin bus clock filter width.

2.39 RNGA: Random Number Generator Accelerator Driver

FSL_RNGA_DRIVER_VERSION

RNGA driver version 2.0.2.

enum *_rnga_mode*

RNGA working mode.

Values:

enumerator kRNGA_ModeNormal

Normal Mode. The ring-oscillator clocks are active; RNGA generates entropy (randomness) from the clocks and stores it in shift registers.

enumerator kRNGA_ModeSleep

Sleep Mode. The ring-oscillator clocks are inactive; RNGA does not generate entropy.

typedef enum *_rnga_mode* rnga_mode_t

RNGA working mode.

void RNGA_Init(RNG_Type *base)

Initializes the RNGA.

This function initializes the RNGA. When called, the RNGA entropy generation starts immediately.

Parameters

- base – RNGA base address

void RNGA_Deinit(RNG_Type *base)

Shuts down the RNGA.

This function shuts down the RNGA.

Parameters

- base – RNGA base address

status_t RNGA_GetRandomData(RNG_Type *base, void *data, size_t data_size)

Gets random data.

This function gets random data from the RNGA.

Parameters

- base – RNGA base address
- data – pointer to user buffer to be filled by random data
- data_size – size of data in bytes

Returns

RNGA status

void RNGA_Seed(RNG_Type *base, uint32_t seed)

Feeds the RNGA module.

This function inputs an entropy value that the RNGA uses to seed its pseudo-random algorithm.

Parameters

- base – RNGA base address
- seed – input seed value

void RNGA_SetMode(RNG_Type *base, *rnga_mode_t* mode)

Sets the RNGA in normal mode or sleep mode.

This function sets the RNGA in sleep mode or normal mode.

Parameters

- base – RNGA base address
- mode – normal mode or sleep mode

rnga_mode_t RNGA_GetMode(RNG_Type *base)

Gets the RNGA working mode.

This function gets the RNGA working mode.

Parameters

- base – RNGA base address

Returns

normal mode or sleep mode

2.40 SIM: System Integration Module Driver

FSL_SIM_DRIVER_VERSION

Driver version.

enum _sim_usb_volt_reg_enable_mode

USB voltage regulator enable setting.

Values:

enumerator kSIM_UsbVoltRegEnable

Enable voltage regulator.

enumerator kSIM_UsbVoltRegEnableInLowPower

Enable voltage regulator in VLPR/VLPW modes.

enumerator kSIM_UsbVoltRegEnableInStop

Enable voltage regulator in STOP/VLPS/LLS/VLLS modes.

enumerator kSIM_UsbVoltRegEnableInAllModes

Enable voltage regulator in all power modes.

enum _sim_flash_mode

Flash enable mode.

Values:

enumerator kSIM_FlashDisableInWait

Disable flash in wait mode.

enumerator kSIM_FlashDisable

Disable flash in normal mode.

typedef struct _sim_uid sim_uid_t

Unique ID.

void SIM_SetUsbVoltRegulatorEnableMode(uint32_t mask)

Sets the USB voltage regulator setting.

This function configures whether the USB voltage regulator is enabled in normal RUN mode, STOP/VLPS/LLS/VLLS modes, and VLPR/VLPW modes. The configurations are passed in as mask value of _sim_usb_volt_reg_enable_mode. For example, to enable USB voltage regulator in RUN/VLPR/VLPW modes and disable in STOP/VLPS/LLS/VLLS mode, use:

```
SIM_SetUsbVoltRegulatorEnableMode(kSIM_UsbVoltRegEnable  
kSIM_UsbVoltRegEnableInLowPower);
```

Parameters

- mask – USB voltage regulator enable setting.

void SIM_GetUniqueId(sim_uid_t *uid)

Gets the unique identification register value.

Parameters

- uid – Pointer to the structure to save the UID value.

```
static inline void SIM_SetFlashMode(uint8_t mode)
```

Sets the flash enable mode.

Parameters

- `mode` – The mode to set; see `_sim_flash_mode` for mode details.

```
struct _sim_uid
```

`#include <fsl_sim.h>` Unique ID.

Public Members

```
uint32_t H
```

UIDH.

```
uint32_t M
```

SIM_UIDM.

```
uint32_t L
```

UIDL.

2.41 SLCD: Segment LCD Driver

```
void SLCD_Init(LCD_Type *base, slcd_config_t *configure)
```

Initializes the SLCD, ungates the module clock, initializes the power setting, enables all used plane pins, and sets with interrupt and work mode with the configuration.

Parameters

- `base` – SLCD peripheral base address.
- `configure` – SLCD configuration pointer. For the configuration structure, many parameters have the default setting and the `SLCD_Getdefaultconfig()` is provided to get them. Use it verified for their applications. The others have no default settings, such as “clk-Config”, and must be provided by the application before calling the `SLCD_Init()` API.

```
void SLCD_Deinit(LCD_Type *base)
```

Deinitializes the SLCD module, gates the module clock, disables an interrupt, and displays the SLCD.

Parameters

- `base` – SLCD peripheral base address.

```
void SLCD_GetDefaultConfig(slcd_config_t *configure)
```

Gets the SLCD default configuration structure. The purpose of this API is to get default parameters of the configuration structure for the `SLCD_Init()`. Use these initialized parameters unchanged in `SLCD_Init()` or modify fields of the structure before the calling `SLCD_Init()`. All default parameters of the configure structuration are listed.

```
config.displayMode      = kSLCD_NormalMode;
config.powerSupply      = kSLCD_InternalVll3UseChargePump;
config.voltageTrim      = kSLCD_RegulatedVolatgeTrim00;
config.lowPowerBehavior = kSLCD_EnabledInWaitStop;
config.interruptSrc     = 0;
config.faultConfig      = NULL;
config.frameFreqIntEnable = false;
```

Parameters

- `configure` – The SLCD configuration structure pointer.

`static inline void SLCD_StartDisplay(LCD_Type *base)`

Enables the SLCD controller, starts generation, and displays the front plane and back plane waveform.

Parameters

- `base` – SLCD peripheral base address.

`static inline void SLCD_StopDisplay(LCD_Type *base)`

Stops the SLCD controller. There is no waveform generator and all enabled pins only output a low value.

Parameters

- `base` – SLCD peripheral base address.

`void SLCD_StartBlinkMode(LCD_Type *base, slcd_blink_mode_t mode, slcd_blink_rate_t rate)`

Starts the SLCD blink mode.

Parameters

- `base` – SLCD peripheral base address.
- `mode` – SLCD blink mode.
- `rate` – SLCD blink rate.

`static inline void SLCD_StopBlinkMode(LCD_Type *base)`

Stops the SLCD blink mode.

Parameters

- `base` – SLCD peripheral base address.

`static inline void SLCD_SetBackPlanePhase(LCD_Type *base, uint32_t pinIndx, slcd_phase_type_t phase)`

Sets the SLCD back plane pin phase.

This function sets the SLCD back plane pin phase. “`kSLCD_PhaseXActivate`” setting means the phase X is active for the back plane pin. “`kSLCD_NoPhaseActivate`” setting means there is no phase active for the back plane pin. For example, set the back plane pin 20 for phase A.

```
SLCD_SetBackPlanePhase(LCD, 20, kSLCD_PhaseAActivate);
```

Parameters

- `base` – SLCD peripheral base address.
- `pinIndx` – SLCD back plane pin index. Range from 0 to 63.
- `phase` – The phase activates for the back plane pin.

`static inline void SLCD_SetFrontPlaneSegments(LCD_Type *base, uint32_t pinIndx, uint8_t operation)`

Sets the SLCD front plane segment operation for a front plane pin.

This function sets the SLCD front plane segment on or off operation. Each bit turns on or off the segments associated with the front plane pin in the following pattern: HGFEDCBA (most significant bit controls segment H and least significant bit controls segment A). For example, turn on the front plane pin 20 for phase B and phase C.

```
SLCD_SetFrontPlaneSegments(LCD, 20, (kSLCD_PhaseBActivate | kSLCD_PhaseCActivate));
```

Parameters

- base – SLCD peripheral base address.
- pinIndx – SLCD back plane pin index. Range from 0 to 63.
- operation – The operation for the segment on the front plane pin. This is a logical OR of the enumeration :: slcd_phase_type_t.

```
static inline void SLCD_SetFrontPlaneOnePhase(LCD_Type *base, uint32_t pinIndx,
                                              slcd_phase_index_t phaseIndx, bool enable)
```

Sets one SLCD front plane pin for one phase.

This function can be used to set one phase on or off for the front plane pin. It can be call many times to set the plane pin for different phase indexes. For example, turn on the front plane pin 20 for phase B and phase C.

```
SLCD_SetFrontPlaneOnePhase(LCD, 20, kSLCD_PhaseBIndex, true);
SLCD_SetFrontPlaneOnePhase(LCD, 20, kSLCD_PhaseCIndex, true);
```

Parameters

- base – SLCD peripheral base address.
- pinIndx – SLCD back plane pin index. Range from 0 to 63.
- phaseIndx – The phase bit index slcd_phase_index_t.
- enable – True to turn on the segment for phaseIndx phase false to turn off the segment for phaseIndx phase.

```
static inline void SLCD_EnablePadSafeState(LCD_Type *base, bool enable)
```

Enables/disables the SLCD pad safe state.

Forces the safe state on the LCD pad controls. All LCD front plane and backplane functions are disabled.

Parameters

- base – SLCD peripheral base address.
- enable – True enable, false disable.

```
static inline uint32_t SLCD_GetFaultDetectCounter(LCD_Type *base)
```

Gets the SLCD fault detect counter.

This function gets the number of samples inside the fault detection sample window.

Parameters

- base – SLCD peripheral base address.

Returns

The fault detect counter. The maximum return value is 255. If the maximum 255 returns, the overflow may happen. Reconfigure the fault detect sample window and fault detect clock prescaler for proper sampling.

```
void SLCD_EnableInterrupts(LCD_Type *base, uint32_t mask)
```

Enables the SLCD interrupt. For example, to enable fault detect complete interrupt and frame frequency interrupt, for FSL_FEATURE_SLCD_HAS_FRAME_FREQUENCY_INTERRUPT enabled case, do the following.

```
SLCD__EnableInterrupts(LCD,kSLCD__FaultDetectCompleteInterrupt | kSLCD__FrameFreqInterrupt);
```

Parameters

- base – SLCD peripheral base address.
- mask – SLCD interrupts to enable. This is a logical OR of the enumeration :: slcd_interrupt_enable_t.

```
void SLCD__DisableInterrupts(LCD_Type *base, uint32_t mask)
```

Disables the SLCD interrupt. For example, to disable fault detect complete interrupt and frame frequency interrupt, for FSL_FEATURE_SLCD_HAS_FRAME_FREQUENCY_INTERRUPT enabled case, do the following.

```
SLCD__DisableInterrupts(LCD,kSLCD__FaultDetectCompleteInterrupt | kSLCD__FrameFreqInterrupt);
```

Parameters

- base – SLCD peripheral base address.
- mask – SLCD interrupts to disable. This is a logical OR of the enumeration :: slcd_interrupt_enable_t.

```
uint32_t SLCD__GetInterruptStatus(LCD_Type *base)
```

Gets the SLCD interrupt status flag.

Parameters

- base – SLCD peripheral base address.

Returns

The event status of the interrupt source. This is the logical OR of members of the enumeration :: slcd_interrupt_enable_t.

```
void SLCD__ClearInterruptStatus(LCD_Type *base, uint32_t mask)
```

Clears the SLCD interrupt events status flag.

Parameters

- base – SLCD peripheral base address.
- mask – SLCD interrupt source to be cleared. This is the logical OR of members of the enumeration :: slcd_interrupt_enable_t.

```
FSL_SLCD_DRIVER_VERSION
```

SLCD driver version.

```
enum _slcd_clock_prescaler
```

SLCD clock prescaler to generate frame frequency.

Values:

```
enumerator kSLCD__ClkPrescaler00
```

Prescaler 0.

```
enumerator kSLCD__ClkPrescaler01
```

Prescaler 1.

```
enumerator kSLCD__ClkPrescaler02
```

Prescaler 2.

```
enumerator kSLCD__ClkPrescaler03
```

Prescaler 3.

enumerator kSLCD_ClkPrescaler04
Prescaler 4.

enumerator kSLCD_ClkPrescaler05
Prescaler 5.

enumerator kSLCD_ClkPrescaler06
Prescaler 6.

enumerator kSLCD_ClkPrescaler07
Prescaler 7.

enum _slcd_blink_rate
SLCD blink rate.

Values:

enumerator kSLCD_BlinkRate00
SLCD blink rate is LCD clock/((2¹²)).

enumerator kSLCD_BlinkRate01
SLCD blink rate is LCD clock/((2¹³)).

enumerator kSLCD_BlinkRate02
SLCD blink rate is LCD clock/((2¹⁴)).

enumerator kSLCD_BlinkRate03
SLCD blink rate is LCD clock/((2¹⁵)).

enumerator kSLCD_BlinkRate04
SLCD blink rate is LCD clock/((2¹⁶)).

enumerator kSLCD_BlinkRate05
SLCD blink rate is LCD clock/((2¹⁷)).

enumerator kSLCD_BlinkRate06
SLCD blink rate is LCD clock/((2¹⁸)).

enumerator kSLCD_BlinkRate07
SLCD blink rate is LCD clock/((2¹⁹)).

enum _slcd_power_supply_option
SLCD power supply option.

Values:

enumerator kSLCD_InternalVll3UseChargePump
VLL3 connected to VDD internally, charge pump is used to generate VLL1 and VLL2.

enumerator kSLCD_ExternalVll3UseResistorBiasNetwork
VLL3 is driven externally and resistor bias network is used to generate VLL1 and VLL2.

enumerator kSLCD_ExternalVll3UseChargePump
VLL3 is driven externally and charge pump is used to generate VLL1 and VLL2.

enumerator kSLCD_InternalVll1UseChargePump
VIREG is connected to VLL1 internally and charge pump is used to generate VLL2 and VLL3.

enum _slcd_regulated_voltage_trim
SLCD regulated voltage trim parameter, be used to meet the desired contrast.

Values:

enumerator kSLCD__RegulatedVolatgeTrim00

Increase the voltage to 0.91 V.

enumerator kSLCD__RegulatedVolatgeTrim01

Increase the voltage to 1.01 V.

enumerator kSLCD__RegulatedVolatgeTrim02

Increase the voltage to 0.96 V.

enumerator kSLCD__RegulatedVolatgeTrim03

Increase the voltage to 1.06 V.

enumerator kSLCD__RegulatedVolatgeTrim04

Increase the voltage to 0.93 V.

enumerator kSLCD__RegulatedVolatgeTrim05

Increase the voltage to 1.03 V.

enumerator kSLCD__RegulatedVolatgeTrim06

Increase the voltage to 0.98 V.

enumerator kSLCD__RegulatedVolatgeTrim07

Increase the voltage to 1.07 V.

enumerator kSLCD__RegulatedVolatgeTrim08

Increase the voltage to 0.92 V.

enumerator kSLCD__RegulatedVolatgeTrim09

Increase the voltage to 1.02 V.

enumerator kSLCD__RegulatedVolatgeTrim10

Increase the voltage to 0.97 V.

enumerator kSLCD__RegulatedVolatgeTrim11

Increase the voltage to 1.08 V.

enumerator kSLCD__RegulatedVolatgeTrim12

Increase the voltage to 0.94 V.

enumerator kSLCD__RegulatedVolatgeTrim13

Increase the voltage to 1.05 V.

enumerator kSLCD__RegulatedVolatgeTrim14

Increase the voltage to 0.99 V.

enumerator kSLCD__RegulatedVolatgeTrim15

Increase the voltage to 1.09 V.

enum _slcd_load_adjust

SLCD load adjust to handle different LCD glass capacitance or configure the LCD charge pump clock source. Adjust the LCD glass capacitance if resistor bias network is enabled: kSLCD_LowLoadOrFastestClkSrc - Low load (LCD glass capacitance 2000pF or lower. LCD or GPIO function can be used on VLL1,VLL2,Vcap1 and Vcap2 pins) kSLCD_LowLoadOrIntermediateClkSrc - low load (LCD glass capacitance 2000pF or lower. LCD or GPIO function can be used on VLL1,VLL2,Vcap1 and Vcap2 pins) kSLCD_HighLoadOrIntermediateClkSrc - high load (LCD glass capacitance 8000pF or lower. LCD or GPIO function can be used on Vcap1 and Vcap2 pins) kSLCD_HighLoadOrSlowestClkSrc - high load (LCD glass capacitance 8000pF or lower LCD or GPIO function can be used on Vcap1 and Vcap2 pins) Adjust clock for charge pump if charge pump is enabled: kSLCD_LowLoadOrFastestClkSrc - Fasten clock source (LCD glass capacitance 8000pF or 4000pF or lower if Fast Frame Rate is set) kSLCD_LowLoadOrIntermediateClkSrc - Intermediate clock source (LCD glass capacitance

4000pF or 2000pF or lower if Fast Frame Rate is set) kSLCD_HighLoadOrIntermediateClkSrc - Intermediate clock source (LCD glass capacitance 2000pF or 1000pF or lower if Fast Frame Rate is set) kSLCD_HighLoadOrSlowestClkSrc - slowest clock source (LCD glass capacitance 1000pF or 500pF or lower if Fast Frame Rate is set)

Values:

enumerator kSLCD__LowLoadOrFastestClkSrc

Adjust in low load or selects fastest clock.

enumerator kSLCD__LowLoadOrIntermediateClkSrc

Adjust in low load or selects intermediate clock.

enumerator kSLCD__HighLoadOrIntermediateClkSrc

Adjust in high load or selects intermediate clock.

enumerator kSLCD__HighLoadOrSlowestClkSrc

Adjust in high load or selects slowest clock.

enum _slcd_clock_src

SLCD clock source.

Values:

enumerator kSLCD_DefaultClk

Select default clock ERCLK32K.

enumerator kSLCD_AlternateClk1

Select alternate clock source 1 : MCGIRCLK.

enumerator kSLCD_AlternateClk2

Select alternate clock source 2 : OSCERCLK.

enum _slcd_alt_clock_div

SLCD alternate clock divider.

Values:

enumerator kSLCD_AltClkDivFactor1

No divide for alternate clock.

enumerator kSLCD_AltClkDivFactor64

Divide alternate clock with factor 64.

enumerator kSLCD_AltClkDivFactor256

Divide alternate clock with factor 256.

enumerator kSLCD_AltClkDivFactor512

Divide alternate clock with factor 512.

enum _slcd_duty_cycle

SLCD duty cycle.

Values:

enumerator kSLCD_1Div1DutyCycle

LCD use 1 BP 1/1 duty cycle.

enumerator kSLCD_1Div2DutyCycle

LCD use 2 BP 1/2 duty cycle.

enumerator kSLCD_1Div3DutyCycle

LCD use 3 BP 1/3 duty cycle.

enumerator kSLCD_1Div4DutyCycle

LCD use 4 BP 1/4 duty cycle.

enumerator kSLCD_1Div5DutyCycle

LCD use 5 BP 1/5 duty cycle.

enumerator kSLCD_1Div6DutyCycle

LCD use 6 BP 1/6 duty cycle.

enumerator kSLCD_1Div7DutyCycle

LCD use 7 BP 1/7 duty cycle.

enumerator kSLCD_1Div8DutyCycle

LCD use 8 BP 1/8 duty cycle.

enum _slcd_phase_type

SLCD segment phase type.

Values:

enumerator kSLCD_NoPhaseActivate

LCD waveform no phase activates.

enumerator kSLCD_PhaseAActivate

LCD waveform phase A activates.

enumerator kSLCD_PhaseBActivate

LCD waveform phase B activates.

enumerator kSLCD_PhaseCAActivate

LCD waveform phase C activates.

enumerator kSLCD_PhaseDAActivate

LCD waveform phase D activates.

enumerator kSLCD_PhaseEActivate

LCD waveform phase E activates.

enumerator kSLCD_PhaseFAActivate

LCD waveform phase F activates.

enumerator kSLCD_PhaseGActivate

LCD waveform phase G activates.

enumerator kSLCD_PhaseHActivate

LCD waveform phase H activates.

enum _slcd_phase_index

SLCD segment phase bit index.

Values:

enumerator kSLCD_PhaseAIndex

LCD phase A bit index.

enumerator kSLCD_PhaseBIndex

LCD phase B bit index.

enumerator kSLCD_PhaseCIndex

LCD phase C bit index.

enumerator kSLCD_PhaseDIndex

LCD phase D bit index.

enumerator kSLCD_PhaseEIndex

LCD phase E bit index.

enumerator kSLCD_PhaseFIndex

LCD phase F bit index.

enumerator kSLCD_PhaseGIndex

LCD phase G bit index.

enumerator kSLCD_PhaseHIndex

LCD phase H bit index.

enum _slcd_display_mode

SLCD display mode.

Values:

enumerator kSLCD_NormalMode

LCD Normal display mode.

enumerator kSLCD_AlternateMode

LCD Alternate display mode. For four back planes or less.

enumerator kSLCD_BlankMode

LCD Blank display mode.

enum _slcd_blink_mode

SLCD blink mode.

Values:

enumerator kSLCD_BlankDisplayBlink

Display blank during the blink period.

enumerator kSLCD_AltDisplayBlink

Display alternate display during the blink period if duty cycle is lower than 5.

enum _slcd_fault_detect_clock_prescaler

SLCD fault detect clock prescaler.

Values:

enumerator kSLCD_FaultSampleFreqDivider1

Fault detect sample clock frequency is 1/1 bus clock.

enumerator kSLCD_FaultSampleFreqDivider2

Fault detect sample clock frequency is 1/2 bus clock.

enumerator kSLCD_FaultSampleFreqDivider4

Fault detect sample clock frequency is 1/4 bus clock.

enumerator kSLCD_FaultSampleFreqDivider8

Fault detect sample clock frequency is 1/8 bus clock.

enumerator kSLCD_FaultSampleFreqDivider16

Fault detect sample clock frequency is 1/16 bus clock.

enumerator kSLCD_FaultSampleFreqDivider32

Fault detect sample clock frequency is 1/32 bus clock.

enumerator kSLCD_FaultSampleFreqDivider64

Fault detect sample clock frequency is 1/64 bus clock.

enumerator kSLCD_FaultSampleFreqDivider128

Fault detect sample clock frequency is 1/128 bus clock.

enum _slcd_fault_detect_sample_window_width

SLCD fault detect sample window width.

Values:

enumerator kSLCD_FaultDetectWindowWidth4SampleClk

Sample window width is 4 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth8SampleClk

Sample window width is 8 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth16SampleClk

Sample window width is 16 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth32SampleClk

Sample window width is 32 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth64SampleClk

Sample window width is 64 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth128SampleClk

Sample window width is 128 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth256SampleClk

Sample window width is 256 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth512SampleClk

Sample window width is 512 sample clock cycles.

enum _slcd_interrupt_enable

SLCD interrupt source.

Values:

enumerator kSLCD_FaultDetectCompleteInterrupt

SLCD fault detection complete interrupt source.

enumerator kSLCD_FrameFreqInterrupt

SLCD frame frequency interrupt source. Not available in all low-power modes.

enum _slcd_lowpower_behavior

SLCD behavior in low power mode.

Values:

enumerator kSLCD_EnabledInWaitStop

SLCD works in wait and stop mode.

enumerator kSLCD_EnabledInWaitOnly

SLCD works in wait mode and is disabled in stop mode.

enumerator kSLCD_EnabledInStopOnly

SLCD works in stop mode and is disabled in wait mode.

enumerator kSLCD_DisabledInWaitStop

SLCD is disabled in stop mode and wait mode.

typedef enum _slcd_clock_prescaler slcd_clock_prescaler_t

SLCD clock prescaler to generate frame frequency.

`typedef enum _slcd_blink_rate slcd_blink_rate_t`
 SLCD blink rate.

`typedef enum _slcd_power_supply_option slcd_power_supply_option_t`
 SLCD power supply option.

`typedef enum _slcd_regulated_voltage_trim slcd_regulated_voltage_trim_t`
 SLCD regulated voltage trim parameter, be used to meet the desired contrast.

`typedef enum _slcd_load_adjust slcd_load_adjust_t`
 SLCD load adjust to handle different LCD glass capacitance or configure the LCD charge pump clock source. Adjust the LCD glass capacitance if resistor bias network is enabled: `kSLCD_LowLoadOrFastestClkSrc` - Low load (LCD glass capacitance 2000pF or lower. LCD or GPIO function can be used on VLL1,VLL2,Vcap1 and Vcap2 pins) `kSLCD_LowLoadOrIntermediateClkSrc` - low load (LCD glass capacitance 2000pF or lower. LCD or GPIO function can be used on VLL1,VLL2,Vcap1 and Vcap2 pins) `kSLCD_HighLoadOrIntermediateClkSrc` - high load (LCD glass capacitance 8000pF or lower. LCD or GPIO function can be used on Vcap1 and Vcap2 pins) `kSLCD_HighLoadOrSlowestClkSrc` - high load (LCD glass capacitance 8000pF or lower LCD or GPIO function can be used on Vcap1 and Vcap2 pins) Adjust clock for charge pump if charge pump is enabled: `kSLCD_LowLoadOrFastestClkSrc` - Fasten clock source (LCD glass capacitance 8000pF or 4000pF or lower if Fast Frame Rate is set) `kSLCD_LowLoadOrIntermediateClkSrc` - Intermediate clock source (LCD glass capacitance 4000pF or 2000pF or lower if Fast Frame Rate is set) `kSLCD_HighLoadOrIntermediateClkSrc` - Intermediate clock source (LCD glass capacitance 2000pF or 1000pF or lower if Fast Frame Rate is set) `kSLCD_HighLoadOrSlowestClkSrc` - slowest clock source (LCD glass capacitance 1000pF or 500pF or lower if Fast Frame Rate is set)

`typedef enum _slcd_clock_src slcd_clock_src_t`
 SLCD clock source.

`typedef enum _slcd_alt_clock_div slcd_alt_clock_div_t`
 SLCD alternate clock divider.

`typedef struct _slcd_clock_config slcd_clock_config_t`
 SLCD clock configuration structure.

`typedef enum _slcd_duty_cycle slcd_duty_cycle_t`
 SLCD duty cycle.

`typedef enum _slcd_phase_type slcd_phase_type_t`
 SLCD segment phase type.

`typedef enum _slcd_phase_index slcd_phase_index_t`
 SLCD segment phase bit index.

`typedef enum _slcd_display_mode slcd_display_mode_t`
 SLCD display mode.

`typedef enum _slcd_blink_mode slcd_blink_mode_t`
 SLCD blink mode.

`typedef enum _slcd_fault_detect_clock_prescaler slcd_fault_detect_clock_prescaler_t`
 SLCD fault detect clock prescaler.

`typedef enum _slcd_fault_detect_sample_window_width`
`slcd_fault_detect_sample_window_width_t`
 SLCD fault detect sample window width.

`typedef enum _slcd_interrupt_enable slcd_interrupt_enable_t`
 SLCD interrupt source.

```
typedef enum _slcd_lowpower_behavior slcd_lowpower_behavior
```

SLCD behavior in low power mode.

```
typedef struct _slcd_fault_detect_config slcd_fault_detect_config_t
```

SLCD fault frame detection configuration structure.

```
typedef struct _slcd_config slcd_config_t
```

SLCD configuration structure.

```
struct _slcd_clock_config
```

#include <fsl_slcd.h> SLCD clock configuration structure.

Public Members

slcd_clock_src_t clkSource

Clock source. “slcd_clock_src_t” is recommended to be used. The SLCD is optimized to operate using a 32.768kHz clock input.

slcd_alt_clock_div_t altClkDivider

The divider to divide the alternate clock used for alternate clock source.

slcd_clock_prescaler_t clkPrescaler

Clock prescaler.

bool fastFrameRateEnable

Fast frame rate enable flag.

```
struct _slcd_fault_detect_config
```

#include <fsl_slcd.h> SLCD fault frame detection configuration structure.

Public Members

bool faultDetectIntEnable

Fault frame detection interrupt enable flag.

bool faultDetectBackPlaneEnable

True means the pin id fault detected is back plane otherwise front plane.

uint8_t faultDetectPinIndex

Fault detected pin id from 0 to 63.

slcd_fault_detect_clock_prescaler_t faultPrescaler

Fault detect clock prescaler.

slcd_fault_detect_sample_window_width_t width

Fault detect sample window width.

```
struct _slcd_config
```

#include <fsl_slcd.h> SLCD configuration structure.

Public Members

slcd_power_supply_option_t powerSupply

Power supply option.

slcd_regulated_voltage_trim_t voltageTrim

Regulated voltage trim used for the internal regulator VIREG to adjust to facilitate contrast control.

slcd_clock_config_t *clkConfig

Clock configure.

slcd_load_adjust_t loadAdjust

Load adjust to handle glass capacitance.

slcd_display_mode_t displayMode

SLCD display mode.

slcd_duty_cycle_t dutyCycle

Duty cycle.

slcd_lowpower_behavior lowPowerBehavior

SLCD behavior in low power mode.

bool frameFreqIntEnable

Frame frequency interrupt enable flag.

uint32_t slcdLowPinEnabled

Setting enabled SLCD pin 0 ~ pin 31. Setting bit n to 1 means enable pin n.

uint32_t slcdHighPinEnabled

Setting enabled SLCD pin 32 ~ pin 63. Setting bit n to 1 means enable pin (n + 32).

uint32_t backPlaneLowPin

Setting back plane pin 0 ~ pin 31. Setting bit n to 1 means setting pin n as back plane. It should never have the same bit setting as the frontPlane Pin.

uint32_t backPlaneHighPin

Setting back plane pin 32 ~ pin 63. Setting bit n to 1 means setting pin (n + 32) as back plane. It should never have the same bit setting as the frontPlane Pin.

slcd_fault_detect_config_t *faultConfig

Fault frame detection configure. If not requirement, set to NULL.

2.42 Smart Card

FSL_SMARTCARD_DRIVER_VERSION

Smart card driver version 2.3.0.

Smart card Error codes.

Values:

enumerator kStatus_SMARTCARD_Success

Transfer ends successfully

enumerator kStatus_SMARTCARD_TxBusy

Transmit in progress

enumerator kStatus_SMARTCARD_RxBusy

Receiving in progress

enumerator kStatus_SMARTCARD_NoTransferInProgress

No transfer in progress

enumerator kStatus_SMARTCARD_Timeout

Transfer ends with time-out

enumerator kStatus_SMARTCARD_Initialized

Smart card driver is already initialized

enumerator kStatus_SMARTCARD_PhyInitialized

Smart card PHY drive is already initialized

enumerator kStatus_SMARTCARD_CardNotActivated

Smart card is not activated

enumerator kStatus_SMARTCARD_InvalidInput

Function called with invalid input arguments

enumerator kStatus_SMARTCARD_OtherError

Some other error occur

enum _smartcard_control

Control codes for the Smart card protocol timers and misc.

Values:

enumerator kSMARTCARD_EnableADT

enumerator kSMARTCARD_DisableADT

enumerator kSMARTCARD_EnableGTV

enumerator kSMARTCARD_DisableGTV

enumerator kSMARTCARD_ResetWWT

enumerator kSMARTCARD_EnableWWT

enumerator kSMARTCARD_DisableWWT

enumerator kSMARTCARD_ResetCWT

enumerator kSMARTCARD_EnableCWT

enumerator kSMARTCARD_DisableCWT

enumerator kSMARTCARD_ResetBWT

enumerator kSMARTCARD_EnableBWT

enumerator kSMARTCARD_DisableBWT

enumerator kSMARTCARD_EnableInitDetect

enumerator kSMARTCARD_EnableAnack

enumerator kSMARTCARD_DisableAnack

enumerator kSMARTCARD_ConfigureBaudrate

enumerator kSMARTCARD_SetupATRMMode

enumerator kSMARTCARD_SetupT0Mode

enumerator kSMARTCARD_SetupT1Mode

enumerator kSMARTCARD_EnableReceiverMode

enumerator kSMARTCARD_DisableReceiverMode

enumerator kSMARTCARD_EnableTransmitterMode

enumerator kSMARTCARD_DisableTransmitterMode

enumerator kSMARTCARD_ResetWaitTimeMultiplier

enum _smartcard_card_voltage_class

Defines Smart card interface voltage class values.

Values:

enumerator kSMARTCARD_VoltageClassUnknown

enumerator kSMARTCARD_VoltageClassA5_0V

enumerator kSMARTCARD_VoltageClassB3_3V

enumerator kSMARTCARD_VoltageClassC1_8V

enum _smartcard_transfer_state

Defines Smart card I/O transfer states.

Values:

enumerator kSMARTCARD_IdleState

enumerator kSMARTCARD_WaitingForTSSState

enumerator kSMARTCARD_InvalidTSDetectedState

enumerator kSMARTCARD_ReceivingState

enumerator kSMARTCARD_TransmittingState

enum _smartcard_reset_type

Defines Smart card reset types.

Values:

enumerator kSMARTCARD_ColdReset

enumerator kSMARTCARD_WarmReset

enumerator kSMARTCARD_NoColdReset

enumerator kSMARTCARD_NoWarmReset

enum _smartcard_transport_type

Defines Smart card transport protocol types.

Values:

enumerator kSMARTCARD_T0Transport

enumerator kSMARTCARD_T1Transport

enum _smartcard_parity_type

Defines Smart card data parity types.

Values:

enumerator kSMARTCARD_EvenParity

enumerator kSMARTCARD_OddParity

enum _smartcard_card_convention

Defines data Convention format.

Values:

enumerator kSMARTCARD_DirectConvention

enumerator kSMARTCARD_InverseConvention

enum *_smartcard_interface_control*

Defines Smart card interface IC control types.

Values:

enumerator kSMARTCARD_InterfaceSetVcc

enumerator kSMARTCARD_InterfaceSetClockToResetDelay

enumerator kSMARTCARD_InterfaceReadStatus

enum *_smartcard_direction*

Defines transfer direction.

Values:

enumerator kSMARTCARD_Receive

enumerator kSMARTCARD_Transmit

typedef enum *_smartcard_control* smartcard_control_t

Control codes for the Smart card protocol timers and misc.

typedef enum *_smartcard_card_voltage_class* smartcard_card_voltage_class_t

Defines Smart card interface voltage class values.

typedef enum *_smartcard_transfer_state* smartcard_transfer_state_t

Defines Smart card I/O transfer states.

typedef enum *_smartcard_reset_type* smartcard_reset_type_t

Defines Smart card reset types.

typedef enum *_smartcard_transport_type* smartcard_transport_type_t

Defines Smart card transport protocol types.

typedef enum *_smartcard_parity_type* smartcard_parity_type_t

Defines Smart card data parity types.

typedef enum *_smartcard_card_convention* smartcard_card_convention_t

Defines data Convention format.

typedef enum *_smartcard_interface_control* smartcard_interface_control_t

Defines Smart card interface IC control types.

typedef enum *_smartcard_direction* smartcard_direction_t

Defines transfer direction.

typedef void (*smartcard_interface_callback_t)(void *smartcardContext, void *param)

Smart card interface interrupt callback function type.

typedef void (*smartcard_transfer_callback_t)(void *smartcardContext, void *param)

Smart card transfer interrupt callback function type.

typedef void (*smartcard_time_delay_t)(uint32_t us)

Time Delay function used to passive waiting using RTOS [us].

typedef struct *_smartcard_card_params* smartcard_card_params_t

Defines card-specific parameters for Smart card driver.

`typedef struct _smartcard_timers_state smartcard_timers_state_t`

Smart card defines the state of the EMV timers in the Smart card driver.

`typedef struct _smartcard_interface_config smartcard_interface_config_t`

Defines user specified configuration of Smart card interface.

`typedef struct _smartcard_xfer smartcard_xfer_t`

Defines user transfer structure used to initialize transfer.

`typedef struct _smartcard_context smartcard_context_t`

Runtime state of the Smart card driver.

`SMARTCARD_INIT_DELAY_CLOCK_CYCLES`

Smart card global define which specify number of clock cycles until initial 'TS' character has to be received.

`SMARTCARD_EMV_ATR_DURATION_ETU`

Smart card global define which specify number of clock cycles during which ATR string has to be received.

`SMARTCARD_TS_DIRECT_CONVENTION`

Smart card specification initial TS character definition of direct convention.

`SMARTCARD_TS_INVERSE_CONVENTION`

Smart card specification initial TS character definition of inverse convention.

`struct _smartcard_card_params`

`#include <fsl_smartcard.h>` Defines card-specific parameters for Smart card driver.

Public Members

`uint16_t Fi`

4 bits Fi - clock rate conversion integer

`uint8_t fMax`

Maximum Smart card frequency in MHz

`uint8_t WI`

8 bits WI - work wait time integer

`uint8_t Di`

4 bits DI - baud rate divisor

`uint8_t BWI`

4 bits BWI - block wait time integer

`uint8_t CWI`

4 bits CWI - character wait time integer

`uint8_t BGI`

4 bits BGI - block guard time integer

`uint8_t GTN`

8 bits GTN - extended guard time integer

`uint8_t IFSC`

Indicates IFSC value of the card

`uint8_t modeNegotiable`

Indicates if the card acts in negotiable or a specific mode.

uint8_t currentD

4 bits DI - current baud rate divisor

uint8_t status

Indicates smart card status

bool t0Indicated

Indicates ff T=0 indicated in TD1 byte

bool t1Indicated

Indicates if T=1 indicated in TD2 byte

bool atrComplete

Indicates whether the ATR received from the card was complete or not

bool atrValid

Indicates whether the ATR received from the card was valid or not

bool present

Indicates if a smart card is present

bool active

Indicates if the smart card is activated

bool faulty

Indicates whether smart card/interface is faulty

smartcard_card_convention_t convention

Card convention, kSMARTCARD_DirectConvention for direct convention, kSMARTCARD_InverseConvention for inverse convention

struct _smartcard_timers_state

#include <fsl_smartcard.h> Smart card defines the state of the EMV timers in the Smart card driver.

Public Members

volatile bool adtExpired

Indicates whether ADT timer expired

volatile bool wwtExpired

Indicates whether WWT timer expired

volatile bool cwtExpired

Indicates whether CWT timer expired

volatile bool bwtExpired

Indicates whether BWT timer expired

volatile bool initCharTimerExpired

Indicates whether reception timer for initialization character (TS) after the RST has expired

struct _smartcard_interface_config

#include <fsl_smartcard.h> Defines user specified configuration of Smart card interface.

Public Members

uint32_t smartCardClock

Smart card interface clock [Hz]

`uint32_t clockToResetDelay`
Indicates clock to RST apply delay [smart card clock cycles]

`uint8_t clockModule`
Smart card clock module number

`uint8_t clockModuleChannel`
Smart card clock module channel number

`uint8_t clockModuleSourceClock`
Smart card clock module source clock [e.g., BusClk]

smartcard_card_voltage_class_t vcc
Smart card voltage class

`uint8_t controlPort`
Smart card PHY control port instance

`uint8_t controlPin`
Smart card PHY control pin instance

`uint8_t irqPort`
Smart card PHY Interrupt port instance

`uint8_t irqPin`
Smart card PHY Interrupt pin instance

`uint8_t resetPort`
Smart card reset port instance

`uint8_t resetPin`
Smart card reset pin instance

`uint8_t vsel0Port`
Smart card PHY Vsel0 control port instance

`uint8_t vsel0Pin`
Smart card PHY Vsel0 control pin instance

`uint8_t vsel1Port`
Smart card PHY Vsel1 control port instance

`uint8_t vsel1Pin`
Smart card PHY Vsel1 control pin instance

`uint8_t dataPort`
Smart card PHY data port instance

`uint8_t dataPin`
Smart card PHY data pin instance

`uint8_t dataPinMux`
Smart card PHY data pin mux option

`uint8_t tsTimerId`
Numerical identifier of the External HW timer for Initial character detection

`struct __smartcard_xfer`
#include <fsl_smartcard.h> Defines user transfer structure used to initialize transfer.

Public Members

smartcard_direction_t direction

Direction of communication. (RX/TX)

uint8_t *buff

The buffer of data.

size_t size

The number of transferred units.

struct __smartcard_context

#include <fsl_smartcard.h> Runtime state of the Smart card driver.

Public Members

void *base

Smart card module base address

smartcard_direction_t direction

Direction of communication. (RX/TX)

uint8_t *xBuff

The buffer of data being transferred.

volatile size_t xSize

The number of bytes to be transferred.

volatile bool xIsBusy

True if there is an active transfer.

uint8_t txFifoEntryCount

Number of data word entries in transmit FIFO.

uint8_t rxFifoThreshold

The max value of the receiver FIFO threshold.

smartcard_interface_callback_t interfaceCallback

Callback to invoke after interface IC raised interrupt.

smartcard_transfer_callback_t transferCallback

Callback to invoke after transfer event occur.

void *interfaceCallbackParam

Interface callback parameter pointer.

void *transferCallbackParam

Transfer callback parameter pointer.

smartcard_time_delay_t timeDelay

Function which handles time delay defined by user or RTOS.

smartcard_reset_type_t resetType

Indicates whether a Cold reset or Warm reset was requested.

smartcard_transport_type_t tType

Indicates current transfer protocol (T0 or T1)

volatile *smartcard_transfer_state_t* transferState

Indicates the current transfer state

smartcard_timers_state_t timersState

Indicates the state of different protocol timers used in driver

smartcard_card_params_t cardParams

Smart card parameters(ATR and current) and interface slots states(ATR and current)

uint8_t IFSD

Indicates the terminal IFSD

smartcard_parity_type_t parity

Indicates current parity even/odd

volatile bool rxtCrossed

Indicates whether RXT thresholds has been crossed

volatile bool txtCrossed

Indicates whether TXT thresholds has been crossed

volatile bool wtxRequested

Indicates whether WTX has been requested or not

volatile bool parityError

Indicates whether a parity error has been detected

uint8_t statusBytes[2]

Used to store Status bytes SW1, SW2 of the last executed card command response

smartcard_interface_config_t interfaceConfig

Smart card interface configuration structure

bool abortTransfer

Used to abort transfer.

2.43 Smart Card UART Driver

void SMARTCARD_UART_GetDefaultConfig(*smartcard_card_params_t* *cardParams)

Fills in the *smartcard_card_params* structure with default values according to the EMV 4.3 specification.

Parameters

- *cardParams* – The configuration structure of type *smartcard_interface_config_t*. Function fill in members: Fi = 372; Di = 1; currentD = 1; WI = 0x0A; GTN = 0x00; with default values.

status_t SMARTCARD_UART_Init(UART_Type *base, *smartcard_context_t* *context, uint32_t srcClock_Hz)

Initializes a UART peripheral for the Smart card/ISO-7816 operation.

This function un-gates the UART clock, initializes the module to EMV default settings, configures the IRQ, enables the module-level interrupt to the core, and initializes the driver context.

Parameters

- *base* – The UART peripheral base address.
- *context* – A pointer to a smart card driver context structure.
- *srcClock_Hz* – Smart card clock generation module source clock.

Returns

An error code or *kStatus_SMARTCARD_Success*.

`void SMARTCARD_UART_Deinit(UART_Type *base)`

This function disables the UART interrupts, disables the transmitter and receiver, and flushes the FIFOs (for modules that support FIFOs) and gates UART clock in SIM.

Parameters

- `base` – The UART peripheral base address.

`int32_t SMARTCARD_UART_GetTransferRemainingBytes(UART_Type *base,
smartcard_context_t *context)`

Returns whether the previous UART transfer has finished.

When performing an async transfer, call this function to ascertain the context of the current transfer: in progress (or busy) or complete (success). If the transfer is still in progress, the user can obtain the number of words that have not been transferred by reading `xSize` of smart card context structure.

Parameters

- `base` – The UART peripheral base address.
- `context` – A pointer to a Smart card driver context structure.

Returns

The number of bytes not transferred.

`status_t SMARTCARD_UART_AbortTransfer(UART_Type *base, smartcard_context_t *context)`

Terminates an asynchronous UART transfer early.

During an async UART transfer, the user can terminate the transfer early if the transfer is still in progress.

Parameters

- `base` – The UART peripheral base address.
- `context` – A pointer to a Smart card driver context structure.

Return values

- `kStatus_SMARTCARD_Success` – The transfer abort was successful.
- `kStatus_SMARTCARD_NoTransmitInProgress` – No transmission is currently in progress.

`status_t SMARTCARD_UART_TransferNonBlocking(UART_Type *base, smartcard_context_t
*context, smartcard_xfer_t *xfer)`

Transfers data using interrupts.

A non-blocking (also known as asynchronous) function means that the function returns immediately after initiating the transfer function. The application has to get the transfer status to see when the transfer is complete. In other words, after calling non-blocking (asynchronous) transfer function, the application must get the transfer status to check if transmit is completed or not.

Parameters

- `base` – The UART peripheral base address.
- `context` – A pointer to a Smart card driver context structure.
- `xfer` – A pointer to Smart card transfer structure where the linked buffers and sizes are stored.

Returns

An error code or `kStatus_SMARTCARD_Success`.

status_t SMARTCARD_UART_Control(UART_Type *base, *smartcard_context_t* *context, *smartcard_control_t* control, uint32_t param)

Controls the UART module per different user requests.

return An *kStatus_SMARTCARD_OtherError* in case of error return *kStatus_SMARTCARD_Success* in success

Parameters

- base – The UART peripheral base address.
- context – A pointer to a smart card driver context structure.
- control – Smart card command type.
- param – Integer value specific to a control command.

void SMARTCARD_UART_IRQHandler(UART_Type *base, *smartcard_context_t* *context)

Interrupt handler for UART.

This handler uses the buffers stored in the *smartcard_context_t* structures to transfer data. The Smart card driver requires this function to call when the UART interrupt occurs.

Parameters

- base – The UART peripheral base address.
- context – A pointer to a Smart card driver context structure.

void SMARTCARD_UART_ErrIRQHandler(UART_Type *base, *smartcard_context_t* *context)

Error interrupt handler for UART.

This function handles error conditions during a transfer.

Parameters

- base – The UART peripheral base address.
- context – A pointer to a Smart card driver context structure.

void SMARTCARD_UART_TSExpiryCallback(UART_Type *base, *smartcard_context_t* *context)

Handles initial TS character timer time-out event.

Parameters

- base – The UART peripheral base address.
- context – A pointer to a Smart card driver context structure.

void smartcard_uart_TimerStart(uint8_t channel, uint32_t time)

Initializes timer specific channel with input period, enable channel interrupt and start counter.

Parameters

- channel – The timer channel.
- time – The time period.

SMARTCARD_EMV_RX_NACK_THRESHOLD

EMV RX NACK interrupt generation threshold.

SMARTCARD_EMV_TX_NACK_THRESHOLD

EMV TX NACK interrupt generation threshold.

SMARTCARD_EMV_RX_TO_TX_GUARD_TIME_T0

EMV TX & RX GUART TIME default value.

SBR_CAL_ADJUST_D1_T0
BRFA_CAL_ADJUST_D1_T0
SBR_CAL_ADJUST_D2_T0
BRFA_CAL_ADJUST_D2_T0
SBR_CAL_ADJUST_D4_T0
BRFA_CAL_ADJUST_D4_T0
SBR_CAL_ADJUST_D1_T1
BRFA_CAL_ADJUST_D1_T1
SBR_CAL_ADJUST_D2_T1
BRFA_CAL_ADJUST_D2_T1
SBR_CAL_ADJUST_D4_T1
BRFA_CAL_ADJUST_D4_T1

2.44 SMC: System Mode Controller Driver

static inline void SMC_GetVersionId(SMC_Type *base, *smc_version_id_t* *versionId)

Gets the SMC version ID.

This function gets the SMC version ID, including major version number, minor version number, and feature specification number.

Parameters

- base – SMC peripheral base address.
- versionId – Pointer to the version ID structure.

void SMC_GetParam(SMC_Type *base, *smc_param_t* *param)

Gets the SMC parameter.

This function gets the SMC parameter including the enabled power modes.

Parameters

- base – SMC peripheral base address.
- param – Pointer to the SMC param structure.

static inline void SMC_SetPowerModeProtection(SMC_Type *base, uint8_t allowedModes)

Configures all power mode protection settings.

This function configures the power mode protection settings for supported power modes in the specified chip family. The available power modes are defined in the *smc_power_mode_protection_t*. This should be done at an early system level initialization stage. See the reference manual for details. This register can only write once after the power reset.

The allowed modes are passed as bit map. For example, to allow LLS and VLLS, use `SMC_SetPowerModeProtection(kSMC_AllowPowerModeVlls | kSMC_AllowPowerModeVlps)`. To allow all modes, use `SMC_SetPowerModeProtection(kSMC_AllowPowerModeAll)`.

Parameters

- `base` – SMC peripheral base address.
- `allowedModes` – Bitmap of the allowed power modes.

`static inline smc_power_state_t SMC_GetPowerModeState(SMC_Type *base)`

Gets the current power mode status.

This function returns the current power mode status. After the application switches the power mode, it should always check the status to check whether it runs into the specified mode or not. The application should check this mode before switching to a different mode. The system requires that only certain modes can switch to other specific modes. See the reference manual for details and the `smc_power_state_t` for information about the power status.

Parameters

- `base` – SMC peripheral base address.

Returns

Current power mode status.

`void SMC_PreEnterStopModes(void)`

Prepares to enter stop modes.

This function should be called before entering STOP/VLPS/LLS/VLLS modes.

`void SMC_PostExitStopModes(void)`

Recovers after wake up from stop modes.

This function should be called after wake up from STOP/VLPS/LLS/VLLS modes. It is used with `SMC_PreEnterStopModes`.

`void SMC_PreEnterWaitModes(void)`

Prepares to enter wait modes.

This function should be called before entering WAIT/VLPW modes.

`void SMC_PostExitWaitModes(void)`

Recovers after wake up from stop modes.

This function should be called after wake up from WAIT/VLPW modes. It is used with `SMC_PreEnterWaitModes`.

`status_t SMC_SetPowerModeRun(SMC_Type *base)`

Configures the system to RUN power mode.

Parameters

- `base` – SMC peripheral base address.

Returns

SMC configuration error code.

`status_t SMC_SetPowerModeHsrn(SMC_Type *base)`

Configures the system to HSRUN power mode.

Parameters

- `base` – SMC peripheral base address.

Returns

SMC configuration error code.

`status_t SMC_SetPowerModeWait(SMC_Type *base)`

Configures the system to WAIT power mode.

Parameters

- `base` – SMC peripheral base address.

Returns

SMC configuration error code.

status_t SMC_SetPowerModeStop(SMC_Type *base, *smc_partial_stop_option_t* option)

Configures the system to Stop power mode.

Parameters

- base – SMC peripheral base address.
- option – Partial Stop mode option.

Returns

SMC configuration error code.

status_t SMC_SetPowerModeVlpr(SMC_Type *base, bool wakeupMode)

Configures the system to VLPR power mode.

Parameters

- base – SMC peripheral base address.
- wakeupMode – Enter Normal Run mode if true, else stay in VLPR mode.

Returns

SMC configuration error code.

status_t SMC_SetPowerModeVlpw(SMC_Type *base)

Configures the system to VLPW power mode.

Parameters

- base – SMC peripheral base address.

Returns

SMC configuration error code.

status_t SMC_SetPowerModeVlps(SMC_Type *base)

Configures the system to VLPS power mode.

Parameters

- base – SMC peripheral base address.

Returns

SMC configuration error code.

status_t SMC_SetPowerModeLls(SMC_Type *base, const *smc_power_mode_lls_config_t* *config)

Configures the system to LLS power mode.

Parameters

- base – SMC peripheral base address.
- config – The LLS power mode configuration structure

Returns

SMC configuration error code.

status_t SMC_SetPowerModeVlls(SMC_Type *base, const *smc_power_mode_vlls_config_t* *config)

Configures the system to VLLS power mode.

Parameters

- base – SMC peripheral base address.
- config – The VLLS power mode configuration structure.

Returns

SMC configuration error code.

FSL_SMC_DRIVER_VERSION

SMC driver version.

enum _smc_power_mode_protection

Power Modes Protection.

Values:

enumerator kSMC_AllowPowerModeVlls

Allow Very-low-leakage Stop Mode.

enumerator kSMC_AllowPowerModeLls

Allow Low-leakage Stop Mode.

enumerator kSMC_AllowPowerModeVlp

Allow Very-Low-power Mode.

enumerator kSMC_AllowPowerModeHsrn

Allow High-speed Run mode.

enumerator kSMC_AllowPowerModeAll

Allow all power mode.

enum _smc_power_state

Power Modes in PMSTAT.

Values:

enumerator kSMC_PowerStateRun

0000_0001 - Current power mode is RUN

enumerator kSMC_PowerStateStop

0000_0010 - Current power mode is STOP

enumerator kSMC_PowerStateVlpr

0000_0100 - Current power mode is VLPR

enumerator kSMC_PowerStateVlpw

0000_1000 - Current power mode is VLPW

enumerator kSMC_PowerStateVlps

0001_0000 - Current power mode is VLPS

enumerator kSMC_PowerStateLls

0010_0000 - Current power mode is LLS

enumerator kSMC_PowerStateVlls

0100_0000 - Current power mode is VLLS

enumerator kSMC_PowerStateHsrn

1000_0000 - Current power mode is HSRUN

enum _smc_run_mode

Run mode definition.

Values:

enumerator kSMC_RunNormal

Normal RUN mode.

enumerator kSMC_RunVlpr

Very-low-power RUN mode.

enumerator kSMC_Hsrun
High-speed Run mode (HSRUN).

enum __smc_stop_mode
Stop mode definition.

Values:

enumerator kSMC_StopNormal
Normal STOP mode.

enumerator kSMC_StopVlps
Very-low-power STOP mode.

enumerator kSMC_StopLls
Low-leakage Stop mode.

enumerator kSMC_StopVlls
Very-low-leakage Stop mode.

enum __smc_stop_submode
VLLS/LLS stop sub mode definition.

Values:

enumerator kSMC_StopSub0
Stop submode 0, for VLLS0/LLS0.

enumerator kSMC_StopSub1
Stop submode 1, for VLLS1/LLS1.

enumerator kSMC_StopSub2
Stop submode 2, for VLLS2/LLS2.

enumerator kSMC_StopSub3
Stop submode 3, for VLLS3/LLS3.

enum __smc_partial_stop_mode
Partial STOP option.

Values:

enumerator kSMC_PartialStop
STOP - Normal Stop mode

enumerator kSMC_PartialStop1
Partial Stop with both system and bus clocks disabled

enumerator kSMC_PartialStop2
Partial Stop with system clock disabled and bus clock enabled

__smc_status, SMC configuration status.

Values:

enumerator kStatus_SMC_StopAbort
Entering Stop mode is abort

typedef enum __smc_power_mode_protection smc_power_mode_protection_t
Power Modes Protection.

typedef enum __smc_power_state smc_power_state_t
Power Modes in PMSTAT.


```
typedef enum _smc_run_mode smc_run_mode_t
    Run mode definition.

typedef enum _smc_stop_mode smc_stop_mode_t
    Stop mode definition.

typedef enum _smc_stop_submode smc_stop_submode_t
    VLLS/LLS stop sub mode definition.

typedef enum _smc_partial_stop_mode smc_partial_stop_option_t
    Partial STOP option.

typedef struct _smc_version_id smc_version_id_t
    IP version ID definition.

typedef struct _smc_param smc_param_t
    IP parameter definition.

typedef struct _smc_power_mode_lls_config smc_power_mode_lls_config_t
    SMC Low-Leakage Stop power mode configuration.

typedef struct _smc_power_mode_vlls_config smc_power_mode_vlls_config_t
    SMC Very Low-Leakage Stop power mode configuration.

struct _smc_version_id
    #include <fsl_smc.h> IP version ID definition.
```

Public Members

```
uint16_t feature
    Feature Specification Number.

uint8_t minor
    Minor version number.

uint8_t major
    Major version number.

struct _smc_param
    #include <fsl_smc.h> IP parameter definition.
```

Public Members

```
bool hsruntimeEnable
    HSRUN mode enable.

bool llsEnable
    LLS mode enable.

bool lls2Enable
    LLS2 mode enable.

bool vlls0Enable
    VLLS0 mode enable.

struct _smc_power_mode_lls_config
    #include <fsl_smc.h> SMC Low-Leakage Stop power mode configuration.
```

Public Members

smc_stop_submode_t subMode
Low-leakage Stop sub-mode

bool enableLpoClock
Enable LPO clock in LLS mode

struct *_smc_power_mode_vlls_config*
#include <fsl_smc.h> SMC Very Low-Leakage Stop power mode configuration.

Public Members

smc_stop_submode_t subMode
Very Low-leakage Stop sub-mode

bool enablePorDetectInVlls0
Enable Power on reset detect in VLLS mode

bool enableRam2InVlls2
Enable RAM2 power in VLLS2

bool enableLpoClock
Enable LPO clock in VLLS mode

2.45 SPI: Serial Peripheral Interface Driver

2.46 SPI DMA Driver

```
void SPI_MasterTransferCreateHandleDMA(SPI_Type *base, spi_dma_handle_t *handle,  
                                       spi_dma_callback_t callback, void *userData,  
                                       dma_handle_t *txHandle, dma_handle_t *rxHandle)
```

Initialize the SPI master DMA handle.

This function initializes the SPI master DMA handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, user need only call this API once to get the initialized handle.

Parameters

- base – SPI peripheral base address.
- handle – SPI handle pointer.
- callback – User callback function called at the end of a transfer.
- userData – User data for callback.
- txHandle – DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
- rxHandle – DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

```
status_t SPI_MasterTransferDMA(SPI_Type *base, spi_dma_handle_t *handle, spi_transfer_t  
                               *xfer)
```

Perform a non-blocking SPI transfer using DMA.

Note: This interface returned immediately after transfer initiates, users should call `SPI_GetTransferStatus` to poll the transfer status to check whether SPI transfer finished.

Parameters

- `base` – SPI peripheral base address.
- `handle` – SPI DMA handle pointer.
- `xfer` – Pointer to dma transfer structure.

Return values

- `kStatus__Success` – Successfully start a transfer.
- `kStatus__InvalidArgument` – Input argument is invalid.
- `kStatus__SPI_Busy` – SPI is not idle, is running another transfer.

`void SPI_MasterTransferAbortDMA(SPI_Type *base, spi_dma_handle_t *handle)`

Abort a SPI transfer using DMA.

Parameters

- `base` – SPI peripheral base address.
- `handle` – SPI DMA handle pointer.

`status_t SPI_MasterTransferGetCountDMA(SPI_Type *base, spi_dma_handle_t *handle, size_t *count)`

Get the transferred bytes for SPI slave DMA.

Parameters

- `base` – SPI peripheral base address.
- `handle` – SPI DMA handle pointer.
- `count` – Transferred bytes.

Return values

- `kStatus__SPI_Success` – Succeed get the transfer count.
- `kStatus__NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

`static inline void SPI_SlaveTransferCreateHandleDMA(SPI_Type *base, spi_dma_handle_t *handle, spi_dma_callback_t callback, void *userData, dma_handle_t *txHandle, dma_handle_t *rxHandle)`

Initialize the SPI slave DMA handle.

This function initializes the SPI slave DMA handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, user need only call this API once to get the initialized handle.

Parameters

- `base` – SPI peripheral base address.
- `handle` – SPI handle pointer.
- `callback` – User callback function called at the end of a transfer.
- `userData` – User data for callback.
- `txHandle` – DMA handle pointer for SPI Tx, the handle shall be static allocated by users.

- rxHandle – DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

```
static inline status_t SPI_SlaveTransferDMA(SPI_Type *base, spi_dma_handle_t *handle,
                                           spi_transfer_t *xfer)
```

Perform a non-blocking SPI transfer using DMA.

Note: This interface returned immediately after transfer initiates, users should call SPI_GetTransferStatus to poll the transfer status to check whether SPI transfer finished.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.
- xfer – Pointer to dma transfer structure.

Return values

- kStatus__Success – Successfully start a transfer.
- kStatus__InvalidArgument – Input argument is invalid.
- kStatus__SPI_Busy – SPI is not idle, is running another transfer.

```
static inline void SPI_SlaveTransferAbortDMA(SPI_Type *base, spi_dma_handle_t *handle)
```

Abort a SPI transfer using DMA.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.

```
static inline status_t SPI_SlaveTransferGetCountDMA(SPI_Type *base, spi_dma_handle_t
                                                    *handle, size_t *count)
```

Get the transferred bytes for SPI slave DMA.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.
- count – Transferred bytes.

Return values

- kStatus__SPI_Success – Succeed get the transfer count.
- kStatus__NoTransferInProgress – There is not a non-blocking transaction currently in progress.

FSL_SPI_DMA_DRIVER_VERSION

SPI DMA driver version.

```
typedef struct _spi_dma_handle spi_dma_handle_t
```

```
typedef void (*spi_dma_callback_t)(SPI_Type *base, spi_dma_handle_t *handle, status_t status,
void *userData)
```

SPI DMA callback called at the end of transfer.

```
struct _spi_dma_handle
```

#include <fsl_spi_dma.h> SPI DMA transfer handle, users should not touch the content of the handle.

Public Members

`bool txInProgress`
Send transfer finished

`bool rxInProgress`
Receive transfer finished

`dma_handle_t *txHandle`
DMA handler for SPI send

`dma_handle_t *rxHandle`
DMA handler for SPI receive

`uint8_t bytesPerFrame`
Bytes in a frame for SPI transfer

`spi_dma_callback_t callback`
Callback for SPI DMA transfer

`void *userData`
User Data for SPI DMA callback

`uint32_t state`
Internal state of SPI DMA transfer

`size_t transferSize`
Bytes need to be transfer

2.47 SPI Driver

`void SPI_MasterGetDefaultConfig(spi_master_config_t *config)`

Sets the SPI master configuration structure to default values.

The purpose of this API is to get the configuration structure initialized for use in `SPI_MasterInit()`. User may use the initialized structure unchanged in `SPI_MasterInit()`, or modify some fields of the structure before calling `SPI_MasterInit()`. After calling this API, the master is ready to transfer. Example:

```
spi_master_config_t config;
SPI_MasterGetDefaultConfig(&config);
```

Parameters

- `config` – pointer to master config structure

`void SPI_MasterInit(SPI_Type *base, const spi_master_config_t *config, uint32_t srcClock_Hz)`

Initializes the SPI with master configuration.

The configuration structure can be filled by user from scratch, or be set with default values by `SPI_MasterGetDefaultConfig()`. After calling this API, the slave is ready to transfer. Example

```
spi_master_config_t config = {
    .baudRate_Bps = 400000,
    ...
};
SPI_MasterInit(SPI0, &config);
```

Parameters

- base – SPI base pointer
- config – pointer to master configuration structure
- srcClock_Hz – Source clock frequency.

void SPI_SlaveGetDefaultConfig(*spi_slave_config_t* *config)

Sets the SPI slave configuration structure to default values.

The purpose of this API is to get the configuration structure initialized for use in SPI_SlaveInit(). Modify some fields of the structure before calling SPI_SlaveInit(). Example:

```
spi_slave_config_t config;  
SPI_SlaveGetDefaultConfig(&config);
```

Parameters

- config – pointer to slave configuration structure

void SPI_SlaveInit(SPI_Type *base, const *spi_slave_config_t* *config)

Initializes the SPI with slave configuration.

The configuration structure can be filled by user from scratch or be set with default values by SPI_SlaveGetDefaultConfig(). After calling this API, the slave is ready to transfer. Example

```
spi_slave_config_t config = {  
    .polarity = kSPIClockPolarity_ActiveHigh;  
    .phase = kSPIClockPhase_FirstEdge;  
    .direction = kSPIMsbFirst;  
    ...  
};  
SPI_MasterInit(SPI0, &config);
```

Parameters

- base – SPI base pointer
- config – pointer to master configuration structure

void SPI_Deinit(SPI_Type *base)

De-initializes the SPI.

Calling this API resets the SPI module, gates the SPI clock. The SPI module can't work unless calling the SPI_MasterInit/SPI_SlaveInit to initialize module.

Parameters

- base – SPI base pointer

static inline void SPI_Enable(SPI_Type *base, bool enable)

Enables or disables the SPI.

Parameters

- base – SPI base pointer
- enable – pass true to enable module, false to disable module

uint32_t SPI_GetStatusFlags(SPI_Type *base)

Gets the status flag.

Parameters

- base – SPI base pointer

Returns

SPI Status, use status flag to AND `_spi_flags` could get the related status.

```
static inline void SPI_ClearInterrupt(SPI_Type *base, uint8_t mask)
```

Clear the interrupt if enable INCTLR.

Parameters

- `base` – SPI base pointer
- `mask` – Interrupt need to be cleared The parameter could be any combination of the following values:
 - `kSPI_RxFullAndModfInterruptEnable`
 - `kSPI_TxEmptyInterruptEnable`
 - `kSPI_MatchInterruptEnable`
 - `kSPI_RxFifoNearFullInterruptEnable`
 - `kSPI_TxFifoNearEmptyInterruptEnable`

```
void SPI_EnableInterrupts(SPI_Type *base, uint32_t mask)
```

Enables the interrupt for the SPI.

Parameters

- `base` – SPI base pointer
- `mask` – SPI interrupt source. The parameter can be any combination of the following values:
 - `kSPI_RxFullAndModfInterruptEnable`
 - `kSPI_TxEmptyInterruptEnable`
 - `kSPI_MatchInterruptEnable`
 - `kSPI_RxFifoNearFullInterruptEnable`
 - `kSPI_TxFifoNearEmptyInterruptEnable`

```
void SPI_DisableInterrupts(SPI_Type *base, uint32_t mask)
```

Disables the interrupt for the SPI.

Parameters

- `base` – SPI base pointer
- `mask` – SPI interrupt source. The parameter can be any combination of the following values:
 - `kSPI_RxFullAndModfInterruptEnable`
 - `kSPI_TxEmptyInterruptEnable`
 - `kSPI_MatchInterruptEnable`
 - `kSPI_RxFifoNearFullInterruptEnable`
 - `kSPI_TxFifoNearEmptyInterruptEnable`

```
static inline void SPI_EnableDMA(SPI_Type *base, uint8_t mask, bool enable)
```

Enables the DMA source for SPI.

Parameters

- `base` – SPI base pointer
- `mask` – SPI DMA source.
- `enable` – True means enable DMA, false means disable DMA

```
static inline uint32_t SPI_GetDataRegisterAddress(SPI_Type *base)
```

Gets the SPI tx/rx data register address.

This API is used to provide a transfer address for the SPI DMA transfer configuration.

Parameters

- base – SPI base pointer

Returns

data register address

```
uint32_t SPI_GetInstance(SPI_Type *base)
```

Get the instance for SPI module.

Parameters

- base – SPI base address

```
static inline void SPI_SetPinMode(SPI_Type *base, spi_pin_mode_t pinMode)
```

Sets the pin mode for transfer.

Parameters

- base – SPI base pointer
- pinMode – pin mode for transfer AND `_spi_pin_mode` could get the related configuration.

```
void SPI_MasterSetBaudRate(SPI_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
```

Sets the baud rate for SPI transfer. This is only used in master.

Parameters

- base – SPI base pointer
- baudRate_Bps – baud rate needed in Hz.
- srcClock_Hz – SPI source clock frequency in Hz.

```
static inline void SPI_SetMatchData(SPI_Type *base, uint32_t matchData)
```

Sets the match data for SPI.

The match data is a hardware comparison value. When the value received in the SPI receive data buffer equals the hardware comparison value, the SPI Match Flag in the S register (S[SPMF]) sets. This can also generate an interrupt if the enable bit sets.

Parameters

- base – SPI base pointer
- matchData – Match data.

```
void SPI_EnableFIFO(SPI_Type *base, bool enable)
```

Enables or disables the FIFO if there is a FIFO.

Parameters

- base – SPI base pointer
- enable – True means enable FIFO, false means disable FIFO.

```
status_t SPI_WriteBlocking(SPI_Type *base, uint8_t *buffer, size_t size)
```

Sends a buffer of data bytes using a blocking method.

Note: This function blocks via polling until all bytes have been sent.

Parameters

- base – SPI base pointer
- buffer – The data bytes to send
- size – The number of data bytes to send

Returns

kStatus_SPI_Timeout The transfer timed out and was aborted.

void SPI_WriteData(SPI_Type *base, uint16_t data)

Writes a data into the SPI data register.

Parameters

- base – SPI base pointer
- data – needs to be write.

uint16_t SPI_ReadData(SPI_Type *base)

Gets a data from the SPI data register.

Parameters

- base – SPI base pointer

Returns

Data in the register.

void SPI_SetDummyData(SPI_Type *base, uint8_t dummyData)

Set up the dummy data.

Parameters

- base – SPI peripheral address.
- dummyData – Data to be transferred when tx buffer is NULL.

void SPI_MasterTransferCreateHandle(SPI_Type *base, *spi_master_handle_t* *handle,
spi_master_callback_t callback, void *userData)

Initializes the SPI master handle.

This function initializes the SPI master handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, call this API once to get the initialized handle.

Parameters

- base – SPI peripheral base address.
- handle – SPI handle pointer.
- callback – Callback function.
- userData – User data.

status_t SPI_MasterTransferBlocking(SPI_Type *base, *spi_transfer_t* *xfer)

Transfers a block of data using a polling method.

Parameters

- base – SPI base pointer
- xfer – pointer to *spi_xfer_config_t* structure

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.

status_t SPI_MasterTransferNonBlocking(SPI_Type *base, *spi_master_handle_t* *handle, *spi_transfer_t* *xfer)

Performs a non-blocking SPI interrupt transfer.

Note: The API immediately returns after transfer initialization is finished. Call SPI_GetStatusIRQ() to get the transfer status.

Note: If SPI transfer data frame size is 16 bits, the transfer size cannot be an odd number.

Parameters

- base – SPI peripheral base address.
- handle – pointer to *spi_master_handle_t* structure which stores the transfer state
- xfer – pointer to *spi_xfer_config_t* structure

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_SPI_Busy – SPI is not idle, is running another transfer.

status_t SPI_MasterTransferGetCount(SPI_Type *base, *spi_master_handle_t* *handle, *size_t* *count)

Gets the bytes of the SPI interrupt transferred.

Parameters

- base – SPI peripheral base address.
- handle – Pointer to SPI transfer handle, this should be a static variable.
- count – Transferred bytes of SPI master.

Return values

- kStatus_SPI_Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

void SPI_MasterTransferAbort(SPI_Type *base, *spi_master_handle_t* *handle)

Aborts an SPI transfer using interrupt.

Parameters

- base – SPI peripheral base address.
- handle – Pointer to SPI transfer handle, this should be a static variable.

void SPI_MasterTransferHandleIRQ(SPI_Type *base, *spi_master_handle_t* *handle)

Interrupts the handler for the SPI.

Parameters

- base – SPI peripheral base address.
- handle – pointer to *spi_master_handle_t* structure which stores the transfer state.

```
void SPI_SlaveTransferCreateHandle(SPI_Type *base, spi_slave_handle_t *handle,  
                                spi_slave_callback_t callback, void *userData)
```

Initializes the SPI slave handle.

This function initializes the SPI slave handle which can be used for other SPI slave transactional APIs. Usually, for a specified SPI instance, call this API once to get the initialized handle.

Parameters

- base – SPI peripheral base address.
- handle – SPI handle pointer.
- callback – Callback function.
- userData – User data.

```
status_t SPI_SlaveTransferNonBlocking(SPI_Type *base, spi_slave_handle_t *handle,  
                                     spi_transfer_t *xfer)
```

Performs a non-blocking SPI slave interrupt transfer.

Note: The API returns immediately after the transfer initialization is finished. Call SPI_GetStatusIRQ() to get the transfer status.

Note: If SPI transfer data frame size is 16 bits, the transfer size cannot be an odd number.

Parameters

- base – SPI peripheral base address.
- handle – pointer to spi_slave_handle_t structure which stores the transfer state
- xfer – pointer to spi_xfer_config_t structure

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_SPI_Busy – SPI is not idle, is running another transfer.

```
static inline status_t SPI_SlaveTransferGetCount(SPI_Type *base, spi_slave_handle_t *handle,  
                                                size_t *count)
```

Gets the bytes of the SPI interrupt transferred.

Parameters

- base – SPI peripheral base address.
- handle – Pointer to SPI transfer handle, this should be a static variable.
- count – Transferred bytes of SPI slave.

Return values

- kStatus_SPI_Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

static inline void SPI_SlaveTransferAbort(SPI_Type *base, spi_slave_handle_t *handle)

Aborts an SPI slave transfer using interrupt.

Parameters

- base – SPI peripheral base address.
- handle – Pointer to SPI transfer handle, this should be a static variable.

void SPI_SlaveTransferHandleIRQ(SPI_Type *base, spi_slave_handle_t *handle)

Interrupts a handler for the SPI slave.

Parameters

- base – SPI peripheral base address.
- handle – pointer to spi_slave_handle_t structure which stores the transfer state

FSL_SPI_DRIVER_VERSION

SPI driver version.

Return status for the SPI driver.

Values:

enumerator kStatus_SPI_Busy
SPI bus is busy

enumerator kStatus_SPI_Idle
SPI is idle

enumerator kStatus_SPI_Error
SPI error

enumerator kStatus_SPI_Timeout
SPI timeout polling status flags.

enum _spi_clock_polarity
SPI clock polarity configuration.

Values:

enumerator kSPI_ClockPolarityActiveHigh
Active-high SPI clock (idles low).

enumerator kSPI_ClockPolarityActiveLow
Active-low SPI clock (idles high).

enum _spi_clock_phase
SPI clock phase configuration.

Values:

enumerator kSPI_ClockPhaseFirstEdge
First edge on SPCK occurs at the middle of the first cycle of a data transfer.

enumerator kSPI_ClockPhaseSecondEdge
First edge on SPCK occurs at the start of the first cycle of a data transfer.

enum _spi_shift_direction
SPI data shifter direction options.

Values:

enumerator kSPI_MsbFirst

Data transfers start with most significant bit.

enumerator kSPI_LsbFirst

Data transfers start with least significant bit.

enum _spi_ss_output_mode

SPI slave select output mode options.

Values:

enumerator kSPI_SlaveSelectAsGpio

Slave select pin configured as GPIO.

enumerator kSPI_SlaveSelectFaultInput

Slave select pin configured for fault detection.

enumerator kSPI_SlaveSelectAutomaticOutput

Slave select pin configured for automatic SPI output.

enum _spi_pin_mode

SPI pin mode options.

Values:

enumerator kSPI_PinModeNormal

Pins operate in normal, single-direction mode.

enumerator kSPI_PinModeInput

Bidirectional mode. Master: MOSI pin is input; Slave: MISO pin is input.

enumerator kSPI_PinModeOutput

Bidirectional mode. Master: MOSI pin is output; Slave: MISO pin is output.

enum _spi_data_bitcount_mode

SPI data length mode options.

Values:

enumerator kSPI_8BitMode

8-bit data transmission mode

enumerator kSPI_16BitMode

16-bit data transmission mode

enum _spi_interrupt_enable

SPI interrupt sources.

Values:

enumerator kSPI_RxFullAndModfInterruptEnable

Receive buffer full (SPRF) and mode fault (MODF) interrupt

enumerator kSPI_TxEmptyInterruptEnable

Transmit buffer empty interrupt

enumerator kSPI_MatchInterruptEnable

Match interrupt

enumerator kSPI_RxFifoNearFullInterruptEnable

Receive FIFO nearly full interrupt

enumerator kSPI_TxFifoNearEmptyInterruptEnable

Transmit FIFO nearly empty interrupt

enum _spi_flags

SPI status flags.

Values:

enumerator kSPI_RxBufferFullFlag

Read buffer full flag

enumerator kSPI_MatchFlag

Match flag

enumerator kSPI_TxBufferEmptyFlag

Transmit buffer empty flag

enumerator kSPI_ModeFaultFlag

Mode fault flag

enumerator kSPI_RxFifoNearFullFlag

Rx FIFO near full

enumerator kSPI_TxFifoNearEmptyFlag

Tx FIFO near empty

enumerator kSPI_TxFifoFullFlag

Tx FIFO full

enumerator kSPI_RxFifoEmptyFlag

Rx FIFO empty

enumerator kSPI_TxFifoError

Tx FIFO error

enumerator kSPI_RxFifoError

Rx FIFO error

enumerator kSPI_TxOverflow

Tx FIFO Overflow

enumerator kSPI_RxOverflow

Rx FIFO Overflow

enum _spi_w1c_interrupt

SPI FIFO write-1-to-clear interrupt flags.

Values:

enumerator kSPI_RxFifoFullClearInterrupt

Receive FIFO full interrupt

enumerator kSPI_TxFifoEmptyClearInterrupt

Transmit FIFO empty interrupt

enumerator kSPI_RxNearFullClearInterrupt

Receive FIFO nearly full interrupt

enumerator kSPI_TxNearEmptyClearInterrupt

Transmit FIFO nearly empty interrupt

enum _spi_txfifo_watermark

SPI TX FIFO watermark settings.

Values:

enumerator `kSPI_TxFifoOneFourthEmpty`
SPI tx watermark at 1/4 FIFO size

enumerator `kSPI_TxFifoOneHalfEmpty`
SPI tx watermark at 1/2 FIFO size

enum `_spi_rxfifo_watermark`
SPI RX FIFO watermark settings.

Values:

enumerator `kSPI_RxFifoThreeFourthsFull`
SPI rx watermark at 3/4 FIFO size

enumerator `kSPI_RxFifoOneHalfFull`
SPI rx watermark at 1/2 FIFO size

enum `_spi_dma_enable_t`
SPI DMA source.

Values:

enumerator `kSPI_TxDmaEnable`
Tx DMA request source

enumerator `kSPI_RxDmaEnable`
Rx DMA request source

enumerator `kSPI_DmaAllEnable`
All DMA request source

typedef enum `_spi_clock_polarity` `spi_clock_polarity_t`
SPI clock polarity configuration.

typedef enum `_spi_clock_phase` `spi_clock_phase_t`
SPI clock phase configuration.

typedef enum `_spi_shift_direction` `spi_shift_direction_t`
SPI data shifter direction options.

typedef enum `_spi_ss_output_mode` `spi_ss_output_mode_t`
SPI slave select output mode options.

typedef enum `_spi_pin_mode` `spi_pin_mode_t`
SPI pin mode options.

typedef enum `_spi_data_bitcount_mode` `spi_data_bitcount_mode_t`
SPI data length mode options.

typedef enum `_spi_w1c_interrupt` `spi_w1c_interrupt_t`
SPI FIFO write-1-to-clear interrupt flags.

typedef enum `_spi_txfifo_watermark` `spi_txfifo_watermark_t`
SPI TX FIFO watermark settings.

typedef enum `_spi_rxfifo_watermark` `spi_rxfifo_watermark_t`
SPI RX FIFO watermark settings.

typedef struct `_spi_master_config` `spi_master_config_t`
SPI master user configure structure.

typedef struct `_spi_slave_config` `spi_slave_config_t`
SPI slave user configure structure.

```
typedef struct _spi_transfer spi_transfer_t
```

SPI transfer structure.

```
typedef struct _spi_master_handle spi_master_handle_t
```

```
typedef spi_master_handle_t spi_slave_handle_t
```

Slave handle is the same with master handle

```
typedef void (*spi_master_callback_t)(SPI_Type *base, spi_master_handle_t *handle, status_t status, void *userData)
```

SPI master callback for finished transmit.

```
typedef void (*spi_slave_callback_t)(SPI_Type *base, spi_slave_handle_t *handle, status_t status, void *userData)
```

SPI master callback for finished transmit.

```
volatile uint8_t g_spiDummyData[]
```

Global variable for dummy data value setting.

```
SPI_DUMMYDATA
```

SPI dummy transfer data, the data is sent while txBuff is NULL.

```
SPI_RETRY_TIMES
```

Retry times for waiting flag.

```
struct _spi_master_config
```

#include <fsl_spi.h> SPI master user configure structure.

Public Members

```
bool enableMaster
```

Enable SPI at initialization time

```
bool enableStopInWaitMode
```

SPI stop in wait mode

```
spi_clock_polarity_t polarity
```

Clock polarity

```
spi_clock_phase_t phase
```

Clock phase

```
spi_shift_direction_t direction
```

MSB or LSB

```
spi_data_bitcount_mode_t dataMode
```

8bit or 16bit mode

```
spi_txfifo_watermark_t txWatermark
```

Tx watermark settings

```
spi_rxfifo_watermark_t rxWatermark
```

Rx watermark settings

```
spi_ss_output_mode_t outputMode
```

SS pin setting

```
spi_pin_mode_t pinMode
```

SPI pin mode select

uint32_t baudRate_Bps
Baud Rate for SPI in Hz

struct __spi_slave_config
#include <fsl_spi.h> SPI slave user configure structure.

Public Members

bool enableSlave
Enable SPI at initialization time

bool enableStopInWaitMode
SPI stop in wait mode

spi_clock_polarity_t polarity
Clock polarity

spi_clock_phase_t phase
Clock phase

spi_shift_direction_t direction
MSB or LSB

spi_data_bitcount_mode_t dataMode
8bit or 16bit mode

spi_txfifo_watermark_t txWatermark
Tx watermark settings

spi_rxfifo_watermark_t rxWatermark
Rx watermark settings

spi_pin_mode_t pinMode
SPI pin mode select

struct __spi_transfer
#include <fsl_spi.h> SPI transfer structure.

Public Members

const uint8_t *txData
Send buffer

uint8_t *rxData
Receive buffer

size_t dataSize
Transfer bytes

uint32_t flags
SPI control flag, useless to SPI.

struct __spi_master_handle
#include <fsl_spi.h> SPI transfer handle structure.

Public Members

`const uint8_t *volatile txData`
Transfer buffer

`uint8_t *volatile rxData`
Receive buffer

`volatile size_t txRemainingBytes`
Send data remaining in bytes

`volatile size_t rxRemainingBytes`
Receive data remaining in bytes

`volatile uint32_t state`
SPI internal state

`size_t transferSize`
Bytes to be transferred

`uint8_t bytePerFrame`
SPI mode, 2bytes or 1byte in a frame

`uint8_t watermark`
Watermark value for SPI transfer

`spi_master_callback_t callback`
SPI callback

`void *userData`
Callback parameter

2.48 SYSPMU: System Memory Protection Unit

`void SYSPMU_Init(SYSPMU_Type *base, const sysmpu_config_t *config)`

Initializes the SYSPMU with the user configuration structure.

This function configures the SYSPMU module with the user-defined configuration.

Parameters

- `base` – SYSPMU peripheral base address.
- `config` – The pointer to the configuration structure.

`void SYSPMU_Deinit(SYSPMU_Type *base)`

Deinitializes the SYSPMU regions.

Parameters

- `base` – SYSPMU peripheral base address.

`static inline void SYSPMU_Enable(SYSPMU_Type *base, bool enable)`

Enables/disables the SYSPMU globally.

Call this API to enable or disable the SYSPMU module.

Parameters

- `base` – SYSPMU peripheral base address.
- `enable` – True enable SYSPMU, false disable SYSPMU.

```
static inline void SYSMPU_RegionEnable(SYSMPU_Type *base, uint32_t number, bool enable)
```

Enables/disables the SYSMPU for a special region.

When SYSMPU is enabled, call this API to disable an unused region of an enabled SYSMPU. Call this API to minimize the power dissipation.

Parameters

- `base` – SYSMPU peripheral base address.
- `number` – SYSMPU region number.
- `enable` – True enable the special region SYSMPU, false disable the special region SYSMPU.

```
void SYSMPU_GetHardwareInfo(SYSMPU_Type *base, sysmpu_hardware_info_t  
                           *hardwareInform)
```

Gets the SYSMPU basic hardware information.

Parameters

- `base` – SYSMPU peripheral base address.
- `hardwareInform` – The pointer to the SYSMPU hardware information structure. See “`sysmpu_hardware_info_t`”.

```
void SYSMPU_SetRegionConfig(SYSMPU_Type *base, const sysmpu_region_config_t  
                           *regionConfig)
```

Sets the SYSMPU region.

Note: Due to the SYSMPU protection, the region number 0 does not allow writes from core to affect the start and end address nor the permissions associated with the debugger. It can only write the permission fields associated with the other masters.

Parameters

- `base` – SYSMPU peripheral base address.
- `regionConfig` – The pointer to the SYSMPU user configuration structure. See “`sysmpu_region_config_t`”.

```
void SYSMPU_SetRegionAddr(SYSMPU_Type *base, uint32_t regionNum, uint32_t startAddr,  
                          uint32_t endAddr)
```

Sets the region start and end address.

Memory region start address. Note: bit0 ~ bit4 is always marked as 0 by SYSMPU. The actual start address by SYSMPU is 0-modulo-32 byte address. Memory region end address. Note: bit0 ~ bit4 always be marked as 1 by SYSMPU. The end address used by the SYSMPU is 31-modulo-32 byte address. Note: Due to the SYSMPU protection, the `startAddr` and `endAddr` can't be changed by the core when `regionNum` is 0.

Parameters

- `base` – SYSMPU peripheral base address.
- `regionNum` – SYSMPU region number. The range is from 0 to `FSL_FEATURE_SYSMPU_DESCRIPTOR_COUNT - 1`.
- `startAddr` – Region start address.
- `endAddr` – Region end address.

```
void SYSMPU_SetRegionRwxMasterAccessRights(SYSMPU_Type *base, uint32_t regionNum,  
                                           uint32_t masterNum, const  
                                           sysmpu_rwxrights_master_access_control_t  
                                           *accessRights)
```

Sets the SYSMPU region access rights for masters with read, write, and execute rights. The SYSMPU access rights depend on two board classifications of bus masters. The privilege rights masters and the normal rights masters. The privilege rights masters have the read, write, and execute access rights. Except the normal read and write rights, the execute rights are also allowed for these masters. The privilege rights masters normally range from bus masters 0 - 3. However, the maximum master number is device-specific. See the “SYSMPU_PRIVILEGED_RIGHTS_MASTER_MAX_INDEX”. The normal rights masters access rights control see “SYSMPU_SetRegionRwMasterAccessRights()”.

Parameters

- *base* – SYSMPU peripheral base address.
- *regionNum* – SYSMPU region number. Should range from 0 to FSL_FEATURE_SYSMPU_DESCRIPTOR_COUNT - 1.
- *masterNum* – SYSMPU bus master number. Should range from 0 to SYSMPU_PRIVILEGED_RIGHTS_MASTER_MAX_INDEX.
- *accessRights* – The pointer to the SYSMPU access rights configuration. See “sysmpu_rwxrights_master_access_control_t”.

bool SYSMPU_GetSlavePortErrorStatus(SYSMPU_Type *base, *sysmpu_slave_t* slaveNum)

Gets the numbers of slave ports where errors occur.

Parameters

- *base* – SYSMPU peripheral base address.
- *slaveNum* – SYSMPU slave port number.

Returns

The slave ports error status. true - error happens in this slave port. false - error didn't happen in this slave port.

void SYSMPU_GetDetailErrorAccessInfo(SYSMPU_Type *base, *sysmpu_slave_t* slaveNum, *sysmpu_access_err_info_t* *errInform)

Gets the SYSMPU detailed error access information.

Parameters

- *base* – SYSMPU peripheral base address.
- *slaveNum* – SYSMPU slave port number.
- *errInform* – The pointer to the SYSMPU access error information. See “sysmpu_access_err_info_t”.

FSL_SYSMPU_DRIVER_VERSION

SYSMPU driver version 2.2.3.

enum *_sysmpu_region_total_num*

Describes the number of SYSMPU regions.

Values:

enumerator kSYSMPU_8Regions

SYSMPU supports 8 regions.

enumerator kSYSMPU_12Regions

SYSMPU supports 12 regions.

enumerator kSYSMPU_16Regions

SYSMPU supports 16 regions.

enum `_sysmpu_slave`

SYSMPU slave port number.

Values:

enumerator `kSYSMPU_Slave0`

SYSMPU slave port 0.

enumerator `kSYSMPU_Slave1`

SYSMPU slave port 1.

enumerator `kSYSMPU_Slave2`

SYSMPU slave port 2.

enumerator `kSYSMPU_Slave3`

SYSMPU slave port 3.

enumerator `kSYSMPU_Slave4`

SYSMPU slave port 4.

enum `_sysmpu_err_access_control`

SYSMPU error access control detail.

Values:

enumerator `kSYSMPU_NoRegionHit`

No region hit error.

enumerator `kSYSMPU_NoneOverlappRegion`

Access single region error.

enumerator `kSYSMPU_OverlappRegion`

Access overlapping region error.

enum `_sysmpu_err_access_type`

SYSMPU error access type.

Values:

enumerator `kSYSMPU_ErrTypeRead`

SYSMPU error access type — read.

enumerator `kSYSMPU_ErrTypeWrite`

SYSMPU error access type — write.

enum `_sysmpu_err_attributes`

SYSMPU access error attributes.

Values:

enumerator `kSYSMPU_InstructionAccessInUserMode`

Access instruction error in user mode.

enumerator `kSYSMPU_DataAccessInUserMode`

Access data error in user mode.

enumerator `kSYSMPU_InstructionAccessInSupervisorMode`

Access instruction error in supervisor mode.

enumerator `kSYSMPU_DataAccessInSupervisorMode`

Access data error in supervisor mode.

enum *_sysmpu_supervisor_access_rights*

SYSMPU access rights in supervisor mode for bus master 0 ~ 3.

Values:

enumerator *kSYSMPU_SupervisorReadWriteExecute*

Read write and execute operations are allowed in supervisor mode.

enumerator *kSYSMPU_SupervisorReadExecute*

Read and execute operations are allowed in supervisor mode.

enumerator *kSYSMPU_SupervisorReadWrite*

Read write operations are allowed in supervisor mode.

enumerator *kSYSMPU_SupervisorEqualToUsermode*

Access permission equal to user mode.

enum *_sysmpu_user_access_rights*

SYSMPU access rights in user mode for bus master 0 ~ 3.

Values:

enumerator *kSYSMPU_UserNoAccessRights*

No access allowed in user mode.

enumerator *kSYSMPU_UserExecute*

Execute operation is allowed in user mode.

enumerator *kSYSMPU_UserWrite*

Write operation is allowed in user mode.

enumerator *kSYSMPU_UserWriteExecute*

Write and execute operations are allowed in user mode.

enumerator *kSYSMPU_UserRead*

Read is allowed in user mode.

enumerator *kSYSMPU_UserReadExecute*

Read and execute operations are allowed in user mode.

enumerator *kSYSMPU_UserReadWrite*

Read and write operations are allowed in user mode.

enumerator *kSYSMPU_UserReadWriteExecute*

Read write and execute operations are allowed in user mode.

typedef enum *_sysmpu_region_total_num* *sysmpu_region_total_num_t*

Describes the number of SYSMPU regions.

typedef enum *_sysmpu_slave* *sysmpu_slave_t*

SYSMPU slave port number.

typedef enum *_sysmpu_err_access_control* *sysmpu_err_access_control_t*

SYSMPU error access control detail.

typedef enum *_sysmpu_err_access_type* *sysmpu_err_access_type_t*

SYSMPU error access type.

typedef enum *_sysmpu_err_attributes* *sysmpu_err_attributes_t*

SYSMPU access error attributes.

typedef enum *_sysmpu_supervisor_access_rights* *sysmpu_supervisor_access_rights_t*

SYSMPU access rights in supervisor mode for bus master 0 ~ 3.

```
typedef enum _sysmpu_user_access_rights sysmpu_user_access_rights_t
    SYSMPU access rights in user mode for bus master 0 ~ 3.
```

```
typedef struct _sysmpu_hardware_info sysmpu_hardware_info_t
    SYSMPU hardware basic information.
```

```
typedef struct _sysmpu_access_err_info sysmpu_access_err_info_t
    SYSMPU detail error access information.
```

```
typedef struct _sysmpu_rwxrights_master_access_control
sysmpu_rwxrights_master_access_control_t
    SYSMPU read/write/execute rights control for bus master 0 ~ 3.
```

```
typedef struct _sysmpu_rwrights_master_access_control
sysmpu_rwrights_master_access_control_t
    SYSMPU read/write access control for bus master 4 ~ 7.
```

```
typedef struct _sysmpu_region_config sysmpu_region_config_t
    SYSMPU region configuration structure.
```

This structure is used to configure the regionNum region. The accessRights1[0] ~ accessRights1[3] are used to configure the bus master 0 ~ 3 with the privilege rights setting. The accessRights2[0] ~ accessRights2[3] are used to configure the high master 4 ~ 7 with the normal read write permission. The master port assignment is the chip configuration. Normally, the core is the master 0, debugger is the master 1. Note that the SYSMPU assigns a priority scheme where the debugger is treated as the highest priority master followed by the core and then all the remaining masters. SYSMPU protection does not allow writes from the core to affect the “regionNum 0” start and end address nor the permissions associated with the debugger. It can only write the permission fields associated with the other masters. This protection guarantees that the debugger always has access to the entire address space and those rights can’t be changed by the core or any other bus master. Prepare the region configuration when regionNum is 0.

```
typedef struct _sysmpu_config sysmpu_config_t
    The configuration structure for the SYSMPU initialization.
    This structure is used when calling the SYSMPU_Init function.
```

```
SYSMPU_MASTER_RWATTRIBUTE_START_PORT
    define the start master port with read and write attributes.
```

```
SYSMPU_REGION_RWXRIGHTS_MASTER_SHIFT(n)
    SYSMPU the bit shift for masters with privilege rights: read write and execute.
```

```
SYSMPU_REGION_RWXRIGHTS_MASTER_MASK(n)
    SYSMPU masters with read, write and execute rights bit mask.
```

```
SYSMPU_REGION_RWXRIGHTS_MASTER_WIDTH
    SYSMPU masters with read, write and execute rights bit width.
```

```
SYSMPU_REGION_RWXRIGHTS_MASTER(n, x)
    SYSMPU masters with read, write and execute rights priority setting.
```

```
SYSMPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n)
    SYSMPU masters with read, write and execute rights process enable bit shift.
```

```
SYSMPU_REGION_RWXRIGHTS_MASTER_PE_MASK(n)
    SYSMPU masters with read, write and execute rights process enable bit mask.
```

```
SYSMPU_REGION_RWXRIGHTS_MASTER_PE(n, x)
    SYSMPU masters with read, write and execute rights process enable setting.
```

`SYSMPU_REGION_RWRIGHTS_MASTER_SHIFT(n)`

SYSMPU masters with normal read write permission bit shift.

`SYSMPU_REGION_RWRIGHTS_MASTER_MASK(n)`

SYSMPU masters with normal read write rights bit mask.

`SYSMPU_REGION_RWRIGHTS_MASTER(n, x)`

SYSMPU masters with normal read write rights priority setting.

`struct __sysmpu_hardware_info`

#include <fsl_sysmpu.h> SYSMPU hardware basic information.

Public Members

`uint8_t hardwareRevisionLevel`

Specifies the SYSMPU's hardware and definition reversion level.

`uint8_t slavePortsNumbers`

Specifies the number of slave ports connected to SYSMPU.

`sysmpu_region_total_num_t regionsNumbers`

Indicates the number of region descriptors implemented.

`struct __sysmpu_access_err_info`

#include <fsl_sysmpu.h> SYSMPU detail error access information.

Public Members

`uint32_t master`

Access error master.

`sysmpu_err_attributes_t attributes`

Access error attributes.

`sysmpu_err_access_type_t accessType`

Access error type.

`sysmpu_err_access_control_t accessControl`

Access error control.

`uint32_t address`

Access error address.

`uint8_t processorIdentification`

Access error processor identification.

`struct __sysmpu_rwxrights_master_access_control`

#include <fsl_sysmpu.h> SYSMPU read/write/execute rights control for bus master 0 ~ 3.

Public Members

`sysmpu_supervisor_access_rights_t superAccessRights`

Master access rights in supervisor mode.

`sysmpu_user_access_rights_t userAccessRights`

Master access rights in user mode.

`bool processIdentifierEnable`

Enables or disables process identifier.


```
struct __sysmpu_rwrights_master_access_control
#include <fsl_sysmpu.h> SYSMPU read/write access control for bus master 4 ~ 7.
```

Public Members

bool writeEnable
Enables or disables write permission.

bool readEnable
Enables or disables read permission.

```
struct __sysmpu_region_config
#include <fsl_sysmpu.h> SYSMPU region configuration structure.
```

This structure is used to configure the regionNum region. The accessRights1[0] ~ accessRights1[3] are used to configure the bus master 0 ~ 3 with the privilege rights setting. The accessRights2[0] ~ accessRights2[3] are used to configure the high master 4 ~ 7 with the normal read write permission. The master port assignment is the chip configuration. Normally, the core is the master 0, debugger is the master 1. Note that the SYSMPU assigns a priority scheme where the debugger is treated as the highest priority master followed by the core and then all the remaining masters. SYSMPU protection does not allow writes from the core to affect the “regionNum 0” start and end address nor the permissions associated with the debugger. It can only write the permission fields associated with the other masters. This protection guarantees that the debugger always has access to the entire address space and those rights can’t be changed by the core or any other bus master. Prepare the region configuration when regionNum is 0.

Public Members

uint32_t regionNum
SYSMPU region number; range form 0 ~ FSL_FEATURE_SYSMPU_DESCRIPTOR_COUNT - 1.

uint32_t startAddress
Memory region start address. Note: bit0 ~ bit4 always be marked as 0 by SYSMPU. The actual start address is 0-modulo-32 byte address.

uint32_t endAddress
Memory region end address. Note: bit0 ~ bit4 always be marked as 1 by SYSMPU. The actual end address is 31-modulo-32 byte address.

sysmpu_rwxrights_master_access_control_t accessRights1[4]
Masters with read, write and execute rights setting.

sysmpu_rwrights_master_access_control_t accessRights2[4]
Masters with normal read write rights setting.

uint8_t processIdentifier
Process identifier used when “processIdentifierEnable” set with true.

uint8_t processIdMask
Process identifier mask. The setting bit will ignore the same bit in process identifier.

```
struct __sysmpu_config
#include <fsl_sysmpu.h> The configuration structure for the SYSMPU initialization.
This structure is used when calling the SYSMPU_Init function.
```

Public Members

sysmpu_region_config_t regionConfig

Region access permission.

struct_sysmpu_config *next

Pointer to the next structure.

2.49 UART: Universal Asynchronous Receiver/Transmitter Driver

2.50 UART DMA Driver

```
void UART__TransferCreateHandleDMA(UART_Type *base, uart_dma_handle_t *handle,  
                                   uart_dma_transfer_callback_t callback, void *userData,  
                                   dma_handle_t *txDmaHandle, dma_handle_t  
                                   *rxDmaHandle)
```

Initializes the UART handle which is used in transactional functions and sets the callback.

Parameters

- base – UART peripheral base address.
- handle – Pointer to the *uart_dma_handle_t* structure.
- callback – UART callback, NULL means no callback.
- userData – User callback function data.
- rxDmaHandle – User requested DMA handle for the RX DMA transfer.
- txDmaHandle – User requested DMA handle for the TX DMA transfer.

```
status_t UART__TransferSendDMA(UART_Type *base, uart_dma_handle_t *handle,  
                               uart_transfer_t *xfer)
```

Sends data using DMA.

This function sends data using DMA. This is non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

- base – UART peripheral base address.
- handle – UART handle pointer.
- xfer – UART DMA transfer structure. See *uart_transfer_t*.

Return values

- kStatus_Success – if succeeded; otherwise failed.
- kStatus_UART_TxBusy – Previous transfer ongoing.
- kStatus_InvalidArgument – Invalid argument.

```
status_t UART__TransferReceiveDMA(UART_Type *base, uart_dma_handle_t *handle,  
                                  uart_transfer_t *xfer)
```

Receives data using DMA.

This function receives data using DMA. This is non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

- `base` – UART peripheral base address.
- `handle` – Pointer to the `uart_dma_handle_t` structure.
- `xfer` – UART DMA transfer structure. See `uart_transfer_t`.

Return values

- `kStatus__Success` – if succeeded; otherwise failed.
- `kStatus__UART__RxBusy` – Previous transfer on going.
- `kStatus__InvalidArgument` – Invalid argument.

`void UART__TransferAbortSendDMA(UART_Type *base, uart_dma_handle_t *handle)`

Aborts the send data using DMA.

This function aborts the sent data using DMA.

Parameters

- `base` – UART peripheral base address.
- `handle` – Pointer to `uart_dma_handle_t` structure.

`void UART__TransferAbortReceiveDMA(UART_Type *base, uart_dma_handle_t *handle)`

Aborts the received data using DMA.

This function abort receive data which using DMA.

Parameters

- `base` – UART peripheral base address.
- `handle` – Pointer to `uart_dma_handle_t` structure.

`status_t UART__TransferGetSendCountDMA(UART_Type *base, uart_dma_handle_t *handle, uint32_t *count)`

Gets the number of bytes written to UART TX register.

This function gets the number of bytes written to UART TX register by DMA.

Parameters

- `base` – UART peripheral base address.
- `handle` – UART handle pointer.
- `count` – Send bytes count.

Return values

- `kStatus__NoTransferInProgress` – No send in progress.
- `kStatus__InvalidArgument` – Parameter is invalid.
- `kStatus__Success` – Get successfully through the parameter `count`;

`status_t UART__TransferGetReceiveCountDMA(UART_Type *base, uart_dma_handle_t *handle, uint32_t *count)`

Gets the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- `base` – UART peripheral base address.
- `handle` – UART handle pointer.
- `count` – Receive bytes count.

Return values

- `kStatus_NoTransferInProgress` – No receive in progress.
- `kStatus_InvalidArgument` – Parameter is invalid.
- `kStatus_Success` – Get successfully through the parameter count;

`void UART_TransferDMAHandleIRQ(UART_Type *base, void *uartDmaHandle)`
UART DMA IRQ handle function.

This function handles the UART transmit complete IRQ request and invoke user callback.

Parameters

- `base` – UART peripheral base address.
- `uartDmaHandle` – UART handle pointer.

`FSL_UART_DMA_DRIVER_VERSION`

UART DMA driver version.

`typedef struct _uart_dma_handle uart_dma_handle_t`

`typedef void (*uart_dma_transfer_callback_t)(UART_Type *base, uart_dma_handle_t *handle, status_t status, void *userData)`

UART transfer callback function.

`struct _uart_dma_handle`

`#include <fsl_uart_dma.h>` UART DMA handle.

Public Members

`UART_Type *base`

UART peripheral base address.

`uart_dma_transfer_callback_t callback`

Callback function.

`void *userData`

UART callback function parameter.

`size_t rxDataSizeAll`

Size of the data to receive.

`size_t txDataSizeAll`

Size of the data to send out.

`dma_handle_t *txDmaHandle`

The DMA TX channel used.

`dma_handle_t *rxDmaHandle`

The DMA RX channel used.

`volatile uint8_t txState`

TX transfer state.

`volatile uint8_t rxState`

RX transfer state

2.51 UART Driver

status_t UART_Init(UART_Type *base, const *uart_config_t* *config, uint32_t srcClock_Hz)

Initializes a UART instance with a user configuration structure and peripheral clock.

This function configures the UART module with the user-defined settings. The user can configure the configuration structure and also get the default configuration by using the UART_GetDefaultConfig() function. The example below shows how to use this API to configure UART.

```
uart_config_t uartConfig;
uartConfig.baudRate_Bps = 115200U;
uartConfig.parityMode = kUART_ParityDisabled;
uartConfig.stopBitCount = kUART_OneStopBit;
uartConfig.txFifoWatermark = 0;
uartConfig.rxFifoWatermark = 1;
UART_Init(UART1, &uartConfig, 20000000U);
```

Parameters

- base – UART peripheral base address.
- config – Pointer to the user-defined configuration structure.
- srcClock_Hz – UART clock source frequency in HZ.

Return values

- kStatus_UART_BaudrateNotSupport – Baudrate is not support in current clock source.
- kStatus_Success – Status UART initialize succeed

void UART_Deinit(UART_Type *base)

Deinitializes a UART instance.

This function waits for TX complete, disables TX and RX, and disables the UART clock.

Parameters

- base – UART peripheral base address.

void UART_GetDefaultConfig(*uart_config_t* *config)

Gets the default configuration structure.

This function initializes the UART configuration structure to a default value. The default values are as follows. `uartConfig->baudRate_Bps = 115200U`; `uartConfig->bitCountPerChar = kUART_8BitsPerChar`; `uartConfig->parityMode = kUART_ParityDisabled`; `uartConfig->stopBitCount = kUART_OneStopBit`; `uartConfig->txFifoWatermark = 0`; `uartConfig->rxFifoWatermark = 1`; `uartConfig->idleType = kUART_IdleTypeStartBit`; `uartConfig->enableTx = false`; `uartConfig->enableRx = false`;

Parameters

- config – Pointer to configuration structure.

status_t UART_SetBaudRate(UART_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)

Sets the UART instance baud rate.

This function configures the UART module baud rate. This function is used to update the UART module baud rate after the UART module is initialized by the UART_Init.

```
UART_SetBaudRate(UART1, 115200U, 20000000U);
```

Parameters

- base – UART peripheral base address.
- baudRate_Bps – UART baudrate to be set.

- srcClock_Hz – UART clock source frequency in Hz.

Return values

- kStatus_UART_BaudrateNotSupport – Baudrate is not support in the current clock source.
- kStatus_Success – Set baudrate succeeded.

void UART_Enable9bitMode(UART_Type *base, bool enable)

Enable 9-bit data mode for UART.

This function set the 9-bit mode for UART module. The 9th bit is not used for parity thus can be modified by user.

Parameters

- base – UART peripheral base address.
- enable – true to enable, false to disable.

static inline void UART_SetMatchAddress(UART_Type *base, uint8_t address1, uint8_t address2)

Set the UART slave address.

This function configures the address for UART module that works as slave in 9-bit data mode. One or two address fields can be configured. When the address field's match enable bit is set, the frame it receives with MSB being 1 is considered as an address frame, otherwise it is considered as data frame. Once the address frame matches one of slave's own addresses, this slave is addressed. This address frame and its following data frames are stored in the receive buffer; otherwise the frames will be discarded. To un-address a slave, just send an address frame with unmatched address.

Note: Any UART instance joined in the multi-slave system can work as slave. The position of the address mark is the same as the parity bit when parity is enabled for 8 bit and 9 bit data formats.

Parameters

- base – UART peripheral base address.
- address1 – UART slave address 1.
- address2 – UART slave address 2.

static inline void UART_EnableMatchAddress(UART_Type *base, bool match1, bool match2)

Enable the UART match address feature.

Parameters

- base – UART peripheral base address.
- match1 – true to enable match address1, false to disable.
- match2 – true to enable match address2, false to disable.

static inline void UART_Set9thTransmitBit(UART_Type *base)

Set UART 9th transmit bit.

Parameters

- base – UART peripheral base address.

static inline void UART_Clear9thTransmitBit(UART_Type *base)

Clear UART 9th transmit bit.

Parameters

- base – UART peripheral base address.

uint32_t UART_GetStatusFlags(UART_Type *base)

Gets UART status flags.

This function gets all UART status flags. The flags are returned as the logical OR value of the enumerators `_uart_flags`. To check a specific status, compare the return value with enumerators in `_uart_flags`. For example, to check whether the TX is empty, do the following.

```
if (kUART_TxDataRegEmptyFlag & UART_GetStatusFlags(UART1))
{
    ...
}
```

Parameters

- base – UART peripheral base address.

Returns

UART status flags which are ORed by the enumerators in the `_uart_flags`.

status_t UART_ClearStatusFlags(UART_Type *base, uint32_t mask)

Clears status flags with the provided mask.

This function clears UART status flags with a provided mask. An automatically cleared flag can't be cleared by this function. These flags can only be cleared or set by hardware. `kUART_TxDataRegEmptyFlag`, `kUART_TransmissionCompleteFlag`, `kUART_RxDataRegFullFlag`, `kUART_RxActiveFlag`, `kUART_NoiseErrorInRxDataRegFlag`, `kUART_ParityErrorInRxDataRegFlag`, `kUART_TxFifoEmptyFlag`, `kUART_RxFifoEmptyFlag`

Note: that this API should be called when the Tx/Rx is idle. Otherwise it has no effect.

Parameters

- base – UART peripheral base address.
- mask – The status flags to be cleared; it is logical OR value of `_uart_flags`.

Return values

- `kStatus_UART_FlagCannotClearManually` – The flag can't be cleared by this function but it is cleared automatically by hardware.
- `kStatus_Success` – Status in the mask is cleared.

void UART_EnableInterrupts(UART_Type *base, uint32_t mask)

Enables UART interrupts according to the provided mask.

This function enables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See `_uart_interrupt_enable`. For example, to enable TX empty interrupt and RX full interrupt, do the following.

```
UART_EnableInterrupts(UART1, kUART_TxDataRegEmptyInterruptEnable | kUART_
↳ RxDataRegFullInterruptEnable);
```

Parameters

- base – UART peripheral base address.
- mask – The interrupts to enable. Logical OR of `_uart_interrupt_enable`.

`void UART_DisableInterrupts(UART_Type *base, uint32_t mask)`

Disables the UART interrupts according to the provided mask.

This function disables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See `_uart_interrupt_enable`. For example, to disable TX empty interrupt and RX full interrupt do the following.

```
UART_DisableInterrupts(UART1, kUART_TxDataRegEmptyInterruptEnable | kUART_
↳ RxDataRegFullInterruptEnable);
```

Parameters

- `base` – UART peripheral base address.
- `mask` – The interrupts to disable. Logical OR of `_uart_interrupt_enable`.

`uint32_t UART_GetEnabledInterrupts(UART_Type *base)`

Gets the enabled UART interrupts.

This function gets the enabled UART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators `_uart_interrupt_enable`. To check a specific interrupts enable status, compare the return value with enumerators in `_uart_interrupt_enable`. For example, to check whether TX empty interrupt is enabled, do the following.

```
uint32_t enabledInterrupts = UART_GetEnabledInterrupts(UART1);

if (kUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)
{
    ...
}
```

Parameters

- `base` – UART peripheral base address.

Returns

UART interrupt flags which are logical OR of the enumerators in `_uart_interrupt_enable`.

`static inline uint32_t UART_GetDataRegisterAddress(UART_Type *base)`

Gets the UART data register address.

This function returns the UART data register address, which is mainly used by DMA/eDMA.

Parameters

- `base` – UART peripheral base address.

Returns

UART data register addresses which are used both by the transmitter and the receiver.

`static inline void UART_EnableTxDMA(UART_Type *base, bool enable)`

Enables or disables the UART transmitter DMA request.

This function enables or disables the transmit data register empty flag, `S1[TDRE]`, to generate the DMA requests.

Parameters

- `base` – UART peripheral base address.
- `enable` – True to enable, false to disable.


```
static inline void UART__EnableRxDMA(UART_Type *base, bool enable)
```

Enables or disables the UART receiver DMA.

This function enables or disables the receiver data register full flag, S1[RDRF], to generate DMA requests.

Parameters

- base – UART peripheral base address.
- enable – True to enable, false to disable.

```
static inline void UART__EnableTx(UART_Type *base, bool enable)
```

Enables or disables the UART transmitter.

This function enables or disables the UART transmitter.

Parameters

- base – UART peripheral base address.
- enable – True to enable, false to disable.

```
static inline void UART__EnableRx(UART_Type *base, bool enable)
```

Enables or disables the UART receiver.

This function enables or disables the UART receiver.

Parameters

- base – UART peripheral base address.
- enable – True to enable, false to disable.

```
static inline void UART__WriteByte(UART_Type *base, uint8_t data)
```

Writes to the TX register.

This function writes data to the TX register directly. The upper layer must ensure that the TX register is empty or TX FIFO has empty room before calling this function.

Parameters

- base – UART peripheral base address.
- data – The byte to write.

```
static inline uint8_t UART__ReadByte(UART_Type *base)
```

Reads the RX register directly.

This function reads data from the RX register directly. The upper layer must ensure that the RX register is full or that the TX FIFO has data before calling this function.

Parameters

- base – UART peripheral base address.

Returns

The byte read from UART data register.

```
static inline uint8_t UART__GetRxFifoCount(UART_Type *base)
```

Gets the rx FIFO data count.

Parameters

- base – UART peripheral base address.

Returns

rx FIFO data count.

```
static inline uint8_t UART_GetTxFifoCount(UART_Type *base)
```

Gets the tx FIFO data count.

Parameters

- base – UART peripheral base address.

Returns

tx FIFO data count.

```
void UART_SendAddress(UART_Type *base, uint8_t address)
```

Transmit an address frame in 9-bit data mode.

Parameters

- base – UART peripheral base address.
- address – UART slave address.

```
status_t UART_WriteBlocking(UART_Type *base, const uint8_t *data, size_t length)
```

Writes to the TX register using a blocking method.

This function polls the TX register, waits for the TX register to be empty or for the TX FIFO to have room and writes data to the TX buffer.

Parameters

- base – UART peripheral base address.
- data – Start address of the data to write.
- length – Size of the data to write.

Return values

- kStatus_UART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully wrote all data.

```
status_t UART_ReadBlocking(UART_Type *base, uint8_t *data, size_t length)
```

Read RX data register using a blocking method.

This function polls the RX register, waits for the RX register to be full or for RX FIFO to have data, and reads data from the TX register.

Parameters

- base – UART peripheral base address.
- data – Start address of the buffer to store the received data.
- length – Size of the buffer.

Return values

- kStatus_UART_RxHardwareOverrun – Receiver overrun occurred while receiving data.
- kStatus_UART_NoiseError – A noise error occurred while receiving data.
- kStatus_UART_FramingError – A framing error occurred while receiving data.
- kStatus_UART_ParityError – A parity error occurred while receiving data.
- kStatus_UART_Timeout – Transmission timed out and was aborted.
- kStatus_Success – Successfully received all data.

```
void UART__TransferCreateHandle(UART_Type *base, uart_handle_t *handle,  
                               uart_transfer_callback_t callback, void *userData)
```

Initializes the UART handle.

This function initializes the UART handle which can be used for other UART transactional APIs. Usually, for a specified UART instance, call this API once to get the initialized handle.

Parameters

- base – UART peripheral base address.
- handle – UART handle pointer.
- callback – The callback function.
- userData – The parameter of the callback function.

```
void UART__TransferStartRingBuffer(UART_Type *base, uart_handle_t *handle, uint8_t  
                                   *ringBuffer, size_t ringBufferSize)
```

Sets up the RX ring buffer.

This function sets up the RX ring buffer to a specific UART handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the UART_TransferReceiveNonBlocking() API. If data is already received in the ring buffer, the user can get the received data from the ring buffer directly.

Note: When using the RX ring buffer, one byte is reserved for internal use. In other words, if ringBufferSize is 32, only 31 bytes are used for saving data.

Parameters

- base – UART peripheral base address.
- handle – UART handle pointer.
- ringBuffer – Start address of the ring buffer for background receiving. Pass NULL to disable the ring buffer.
- ringBufferSize – Size of the ring buffer.

```
void UART__TransferStopRingBuffer(UART_Type *base, uart_handle_t *handle)
```

Aborts the background transfer and uninstalls the ring buffer.

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

- base – UART peripheral base address.
- handle – UART handle pointer.

```
size_t UART__TransferGetRxRingBufferLength(uart_handle_t *handle)
```

Get the length of received data in RX ring buffer.

Parameters

- handle – UART handle pointer.

Returns

Length of received data in RX ring buffer.

```
status_t UART__TransferSendNonBlocking(UART_Type *base, uart_handle_t *handle,  
                                       uart_transfer_t *xfer)
```

Transmits a buffer of data using the interrupt method.

This function sends data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data to be written to the TX register. When all data is written to the TX register in the ISR, the UART driver calls the callback function and passes the `kStatus_UART_TxIdle` as status parameter.

Note: The `kStatus_UART_TxIdle` is passed to the upper layer when all data is written to the TX register. However, it does not ensure that all data is sent out. Before disabling the TX, check the `kUART_TransmissionCompleteFlag` to ensure that the TX is finished.

Parameters

- `base` – UART peripheral base address.
- `handle` – UART handle pointer.
- `xfer` – UART transfer structure. See `uart_transfer_t`.

Return values

- `kStatus_Success` – Successfully start the data transmission.
- `kStatus_UART_TxBusy` – Previous transmission still not finished; data not all written to TX register yet.
- `kStatus_InvalidArgument` – Invalid argument.

`void UART_TransferAbortSend(UART_Type *base, uart_handle_t *handle)`

Aborts the interrupt-driven data transmit.

This function aborts the interrupt-driven data sending. The user can get the `remainBytes` to find out how many bytes are not sent out.

Parameters

- `base` – UART peripheral base address.
- `handle` – UART handle pointer.

`status_t UART_TransferGetSendCount(UART_Type *base, uart_handle_t *handle, uint32_t *count)`

Gets the number of bytes sent out to bus.

This function gets the number of bytes sent out to bus by using the interrupt method.

Parameters

- `base` – UART peripheral base address.
- `handle` – UART handle pointer.
- `count` – Send bytes count.

Return values

- `kStatus_NoTransferInProgress` – No send in progress.
- `kStatus_InvalidArgument` – The parameter is invalid.
- `kStatus_Success` – Get successfully through the parameter `count`;

`status_t UART_TransferReceiveNonBlocking(UART_Type *base, uart_handle_t *handle, uart_transfer_t *xfer, size_t *receivedBytes)`

Receives a buffer of data using an interrupt method.

This function receives data using an interrupt method. This is a non-blocking function, which returns without waiting for all data to be received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter `receivedBytes` shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is

not enough to read, the receive request is saved by the UART driver. When the new data arrives, the receive request is serviced first. When all data is received, the UART driver notifies the upper layer through a callback function and passes the status parameter `kStatus_UART_RxIdle`. For example, the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer. The 5 bytes are copied to the `xfer->data` and this function returns with the parameter `receivedBytes` set to 5. For the left 5 bytes, newly arrived data is saved from the `xfer->data[5]`. When 5 bytes are received, the UART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to the `xfer->data`. When all data is received, the upper layer is notified.

Parameters

- `base` – UART peripheral base address.
- `handle` – UART handle pointer.
- `xfer` – UART transfer structure, see `uart_transfer_t`.
- `receivedBytes` – Bytes received from the ring buffer directly.

Return values

- `kStatus_Success` – Successfully queue the transfer into transmit queue.
- `kStatus_UART_RxBusy` – Previous receive request is not finished.
- `kStatus_InvalidArgument` – Invalid argument.

`void UART_TransferAbortReceive(UART_Type *base, uart_handle_t *handle)`

Aborts the interrupt-driven data receiving.

This function aborts the interrupt-driven data receiving. The user can get the `remainBytes` to know how many bytes are not received yet.

Parameters

- `base` – UART peripheral base address.
- `handle` – UART handle pointer.

`status_t UART_TransferGetReceiveCount(UART_Type *base, uart_handle_t *handle, uint32_t *count)`

Gets the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- `base` – UART peripheral base address.
- `handle` – UART handle pointer.
- `count` – Receive bytes count.

Return values

- `kStatus_NoTransferInProgress` – No receive in progress.
- `kStatus_InvalidArgument` – Parameter is invalid.
- `kStatus_Success` – Get successfully through the parameter `count`;

`status_t UART_EnableTxFIFO(UART_Type *base, bool enable)`

Enables or disables the UART Tx FIFO.

This function enables or disables the UART Tx FIFO.

param `base` UART peripheral base address. param `enable` true to enable, false to disable. retval `kStatus_Success` Successfully turn on or turn off Tx FIFO. retval `kStatus_Fail` Fail to turn on or turn off Tx FIFO.

status_t UART_EnableRxFIFO(UART_Type *base, bool enable)

Enables or disables the UART Rx FIFO.

This function enables or disables the UART Rx FIFO.

param base UART peripheral base address. param enable true to enable, false to disable. retval kStatus_Success Successfully turn on or turn off Rx FIFO. retval kStatus_Fail Fail to turn on or turn off Rx FIFO.

static inline void UART_SetRxFifoWatermark(UART_Type *base, uint8_t water)

Sets the rx FIFO watermark.

Parameters

- base – UART peripheral base address.
- water – Rx FIFO watermark.

static inline void UART_SetTxFifoWatermark(UART_Type *base, uint8_t water)

Sets the tx FIFO watermark.

Parameters

- base – UART peripheral base address.
- water – Tx FIFO watermark.

void UART_TransferHandleIRQ(UART_Type *base, void *irqHandle)

UART IRQ handle function.

This function handles the UART transmit and receive IRQ request.

Parameters

- base – UART peripheral base address.
- irqHandle – UART handle pointer.

void UART_TransferHandleErrorIRQ(UART_Type *base, void *irqHandle)

UART Error IRQ handle function.

This function handles the UART error IRQ request.

Parameters

- base – UART peripheral base address.
- irqHandle – UART handle pointer.

FSL_UART_DRIVER_VERSION

UART driver version.

Error codes for the UART driver.

Values:

enumerator kStatus_UART_TxBusy

Transmitter is busy.

enumerator kStatus_UART_RxBusy

Receiver is busy.

enumerator kStatus_UART_TxIdle

UART transmitter is idle.

enumerator kStatus_UART_RxIdle

UART receiver is idle.

enumerator kStatus_UART_TxWatermarkTooLarge
TX FIFO watermark too large

enumerator kStatus_UART_RxWatermarkTooLarge
RX FIFO watermark too large

enumerator kStatus_UART_FlagCannotClearManually
UART flag can't be manually cleared.

enumerator kStatus_UART_Error
Error happens on UART.

enumerator kStatus_UART_RxRingBufferOverrun
UART RX software ring buffer overrun.

enumerator kStatus_UART_RxHardwareOverrun
UART RX receiver overrun.

enumerator kStatus_UART_NoiseError
UART noise error.

enumerator kStatus_UART_FramingError
UART framing error.

enumerator kStatus_UART_ParityError
UART parity error.

enumerator kStatus_UART_BaudrateNotSupport
Baudrate is not support in current clock source

enumerator kStatus_UART_IdleLineDetected
UART IDLE line detected.

enumerator kStatus_UART_Timeout
UART times out.

enum __uart_parity_mode
UART parity mode.

Values:

enumerator kUART_ParityDisabled
Parity disabled

enumerator kUART_ParityEven
Parity enabled, type even, bit setting: PE|PT = 10

enumerator kUART_ParityOdd
Parity enabled, type odd, bit setting: PE|PT = 11

enum __uart_stop_bit_count
UART stop bit count.

Values:

enumerator kUART_OneStopBit
One stop bit

enumerator kUART_TwoStopBit
Two stop bits

enum __uart_idle_type_select
UART idle type select.

Values:

enumerator kUART_IdleTypeStartBit
Start counting after a valid start bit.

enumerator kUART_IdleTypeStopBit
Start counting after a stop bit.

enum __uart_interrupt_enable

UART interrupt configuration structure, default settings all disabled.

This structure contains the settings for all of the UART interrupt configurations.

Values:

enumerator kUART_LinBreakInterruptEnable
LIN break detect interrupt.

enumerator kUART_RxActiveEdgeInterruptEnable
RX active edge interrupt.

enumerator kUART_TxDataRegEmptyInterruptEnable
Transmit data register empty interrupt.

enumerator kUART_TransmissionCompleteInterruptEnable
Transmission complete interrupt.

enumerator kUART_RxDataRegFullInterruptEnable
Receiver data register full interrupt.

enumerator kUART_IdleLineInterruptEnable
Idle line interrupt.

enumerator kUART_RxOverrunInterruptEnable
Receiver overrun interrupt.

enumerator kUART_NoiseErrorInterruptEnable
Noise error flag interrupt.

enumerator kUART_FramingErrorInterruptEnable
Framing error flag interrupt.

enumerator kUART_ParityErrorInterruptEnable
Parity error flag interrupt.

enumerator kUART_RxFifoOverflowInterruptEnable
RX FIFO overflow interrupt.

enumerator kUART_TxFifoOverflowInterruptEnable
TX FIFO overflow interrupt.

enumerator kUART_RxFifoUnderflowInterruptEnable
RX FIFO underflow interrupt.

enumerator kUART_AllInterruptsEnable

UART status flags.

This provides constants for the UART status flags for use in the UART functions.

Values:

enumerator kUART_TxDataRegEmptyFlag
TX data register empty flag.

enumerator `kUART_TransmissionCompleteFlag`
Transmission complete flag.

enumerator `kUART_RxDataRegFullFlag`
RX data register full flag.

enumerator `kUART_IdleLineFlag`
Idle line detect flag.

enumerator `kUART_RxOverrunFlag`
RX overrun flag.

enumerator `kUART_NoiseErrorFlag`
RX takes 3 samples of each received bit. If any of these samples differ, noise flag sets

enumerator `kUART_FramingErrorFlag`
Frame error flag, sets if logic 0 was detected where stop bit expected

enumerator `kUART_ParityErrorFlag`
If parity enabled, sets upon parity error detection

enumerator `kUART_LinBreakFlag`
LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled

enumerator `kUART_RxActiveEdgeFlag`
RX pin active edge interrupt flag, sets when active edge detected

enumerator `kUART_RxActiveFlag`
Receiver Active Flag (RAF), sets at beginning of valid start bit

enumerator `kUART_NoiseErrorInRxDataRegFlag`
Noisy bit, sets if noise detected.

enumerator `kUART_ParityErrorInRxDataRegFlag`
Parity bit, sets if parity error detected.

enumerator `kUART_TxFifoEmptyFlag`
TXEMPT bit, sets if TX buffer is empty

enumerator `kUART_RxFifoEmptyFlag`
RXEMPT bit, sets if RX buffer is empty

enumerator `kUART_TxFifoOverflowFlag`
TXOF bit, sets if TX buffer overflow occurred

enumerator `kUART_RxFifoOverflowFlag`
RXOF bit, sets if receive buffer overflow

enumerator `kUART_RxFifoUnderflowFlag`
RXUF bit, sets if receive buffer underflow

typedef enum *uart_parity_mode* `uart_parity_mode_t`
UART parity mode.

typedef enum *uart_stop_bit_count* `uart_stop_bit_count_t`
UART stop bit count.

typedef enum *uart_idle_type_select* `uart_idle_type_select_t`
UART idle type select.

typedef struct *uart_config* `uart_config_t`
UART configuration structure.

```
typedef struct _uart_transfer uart_transfer_t
```

UART transfer structure.

```
typedef struct _uart_handle uart_handle_t
```

```
typedef void (*uart_transfer_callback_t)(UART_Type *base, uart_handle_t *handle, status_t status, void *userData)
```

UART transfer callback function.

```
typedef void (*uart_isr_t)(UART_Type *base, void *handle)
```

```
void *s_uartHandle[]
```

Pointers to uart handles for each instance.

```
const IRQn_Type s_uartIRQ[]
```

```
uart_isr_t s_uartIsr
```

Pointer to uart IRQ handler for each instance.

```
uint32_t UART_GetInstance(UART_Type *base)
```

Get the UART instance from peripheral base address.

Parameters

- base – UART peripheral base address.

Returns

UART instance.

```
UART_RETRY_TIMES
```

Retry times for waiting flag.

```
struct _uart_config
```

#include <fsl_uart.h> UART configuration structure.

Public Members

```
uint32_t baudRate_Bps
```

UART baud rate

```
uart_parity_mode_t parityMode
```

Parity mode, disabled (default), even, odd

```
uart_stop_bit_count_t stopBitCount
```

Number of stop bits, 1 stop bit (default) or 2 stop bits

```
uint8_t txFifoWatermark
```

TX FIFO watermark

```
uint8_t rxFifoWatermark
```

RX FIFO watermark

```
bool enableRxRTS
```

RX RTS enable

```
bool enableTxCTS
```

TX CTS enable

```
uart_idle_type_select_t idleType
```

IDLE type select.

```
bool enableTx
```

Enable TX

bool enableRx
 Enable RX

struct __uart__transfer
 #include <fsl_uart.h> UART transfer structure.

Public Members

size_t dataSize
 The byte count to be transfer.

struct __uart__handle
 #include <fsl_uart.h> UART handle structure.

Public Members

const uint8_t *volatile txData
 Address of remaining data to send.

volatile size_t txDataSize
 Size of the remaining data to send.

size_t txDataSizeAll
 Size of the data to send out.

uint8_t *volatile rxData
 Address of remaining data to receive.

volatile size_t rxDataSize
 Size of the remaining data to receive.

size_t rxDataSizeAll
 Size of the data to receive.

uint8_t *rxRingBuffer
 Start address of the receiver ring buffer.

size_t rxRingBufferSize
 Size of the ring buffer.

volatile uint16_t rxRingBufferHead
 Index for the driver to store received data into ring buffer.

volatile uint16_t rxRingBufferTail
 Index for the user to get data from the ring buffer.

uart_transfer_callback_t callback
 Callback function.

void *userData
 UART callback function parameter.

volatile uint8_t txState
 TX transfer state.

volatile uint8_t rxState
 RX transfer state

union __unnamed36__

Public Members

uint8_t *data

The buffer of data to be transfer.

uint8_t *rxData

The buffer to receive data.

const uint8_t *txData

The buffer of data to be sent.

2.52 VREF: Voltage Reference Driver

status_t VREF_Init(VREF_Type *base, const *vref_config_t* *config)

Enables the clock gate and configures the VREF module according to the configuration structure.

This function must be called before calling all other VREF driver functions, read/write registers, and configurations with user-defined settings. The example below shows how to set up *vref_config_t* parameters and how to call the VREF_Init function by passing in these parameters. This is an example.

```
vref_config_t vrefConfig;
vrefConfig.bufferMode = kVREF_ModeHighPowerBuffer;
vrefConfig.enableExternalVoltRef = false;
vrefConfig.enableLowRef = false;
VREF_Init(VREF, &vrefConfig);
```

Parameters

- base – VREF peripheral address.
- config – Pointer to the configuration structure.

Return values

- kStatus_Success – run success.
- kStatus_Timeout – timeout occurs.

void VREF_Deinit(VREF_Type *base)

Stops and disables the clock for the VREF module.

This function should be called to shut down the module. This is an example.

```
vref_config_t vrefUserConfig;
VREF_Init(VREF);
VREF_GetDefaultConfig(&vrefUserConfig);
...
VREF_Deinit(VREF);
```

Parameters

- base – VREF peripheral address.

void VREF_GetDefaultConfig(*vref_config_t* *config)

Initializes the VREF configuration structure.

This function initializes the VREF configuration structure to default values. This is an example.

```
vrefConfig->bufferMode = kVREF_ModeHighPowerBuffer;
vrefConfig->enableExternalVoltRef = false;
vrefConfig->enableLowRef = false;
```

Parameters

- config – Pointer to the initialization structure.

status_t VREF_SetTrimVal(VREF_Type *base, uint8_t trimValue)

Sets a TRIM value for the reference voltage.

This function sets a TRIM value for the reference voltage. Note that the TRIM value maximum is 0x3F.

Parameters

- base – VREF peripheral address.
- trimValue – Value of the trim register to set the output reference voltage (maximum 0x3F (6-bit)).

Return values

- kStatus__Success – run success.
- kStatus__Timeout – timeout occurs.

static inline uint8_t VREF_GetTrimVal(VREF_Type *base)

Reads the value of the TRIM meaning output voltage.

This function gets the TRIM value from the TRM register.

Parameters

- base – VREF peripheral address.

Returns

Six-bit value of trim setting.

status_t VREF_SetLowReferenceTrimVal(VREF_Type *base, uint8_t trimValue)

Sets the TRIM value for the low voltage reference.

This function sets the TRIM value for low reference voltage. Note the following.

- The TRIM value maximum is 0x05U
- The values 111b and 110b are not valid/allowed.

Parameters

- base – VREF peripheral address.
- trimValue – Value of the trim register to set output low reference voltage (maximum 0x05U (3-bit)).

Return values

- kStatus__Success – run success.
- kStatus__Timeout – timeout occurs.

static inline uint8_t VREF_GetLowReferenceTrimVal(VREF_Type *base)

Reads the value of the TRIM meaning output voltage.

This function gets the TRIM value from the VREFL_TRM register.

Parameters

- base – VREF peripheral address.

Returns

Three-bit value of the trim setting.

FSL_VREF_DRIVER_VERSION

Version 2.1.3.

VREF_INTERNAL_VOLTAGE_STABLE_TIMEOUT

Max loops to wait for VREF internal voltage stable.

This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

enum _vref_buffer_mode

VREF modes.

Values:

enumerator kVREF_ModeBandgapOnly

Bandgap on only, for stabilization and startup

enumerator kVREF_ModeHighPowerBuffer

High-power buffer mode enabled

enumerator kVREF_ModeLowPowerBuffer

Low-power buffer mode enabled

typedef enum _vref_buffer_mode vref_buffer_mode_t

VREF modes.

typedef struct _vref_config vref_config_t

The description structure for the VREF module.

VREF_SC_MODE_LV

VREF_SC_REGEN

VREF_SC_VREFEN

VREF_SC_ICOMPEN

VREF_SC_REGEN_MASK

VREF_SC_VREFST_MASK

VREF_SC_VREFEN_MASK

VREF_SC_MODE_LV_MASK

VREF_SC_ICOMPEN_MASK

TRM

VREF_TRM_TRIM

VREF_TRM_CHOPEN_MASK

VREF_TRM_TRIM_MASK

VREF_TRM_CHOPEN_SHIFT

VREF_TRM_TRIM_SHIFT

VREF_SC_MODE_LV_SHIFT

VREF_SC_REGEN_SHIFT

VREF_SC_VREFST_SHIFT

VREF_SC_ICOMPEN_SHIFT

struct `_vref_config`

#include <fsl_vref.h> The description structure for the VREF module.

Public Members

`vref_buffer_mode_t` `bufferMode`

Buffer mode selection

bool `enableLowRef`

Set VREFL (0.4 V) reference buffer enable or disable

bool `enableExternalVoltRef`

Select external voltage reference or not (internal)

2.53 WDOG: Watchdog Timer Driver

void `WDOG_GetDefaultConfig(wdog_config_t *config)`

Initializes the WDOG configuration structure.

This function initializes the WDOG configuration structure to default values. The default values are as follows.

```
wdogConfig->enableWdog = true;
wdogConfig->clockSource = kWDOG_LpoClockSource;
wdogConfig->prescaler = kWDOG_ClockPrescalerDivide1;
wdogConfig->workMode.enableWait = true;
wdogConfig->workMode.enableStop = false;
wdogConfig->workMode.enableDebug = false;
wdogConfig->enableUpdate = true;
wdogConfig->enableInterrupt = false;
wdogConfig->enableWindowMode = false;
wdogConfig->windowValue = 0;
wdogConfig->timeoutValue = 0xFFFFU;
```

See also:

`wdog_config_t`

Parameters

- `config` – Pointer to the WDOG configuration structure.

void `WDOG_Init(WDOG_Type *base, const wdog_config_t *config)`

Initializes the WDOG.

This function initializes the WDOG. When called, the WDOG runs according to the configuration. To reconfigure WDOG without forcing a reset first, `enableUpdate` must be set to true in the configuration.

This is an example.

```
wdog_config_t config;  
WDOG_GetDefaultConfig(&config);  
config.timeoutValue = 0x7ffU;  
config.enableUpdate = true;  
WDOG_Init(wdog_base,&config);
```

Parameters

- base – WDOG peripheral base address
- config – The configuration of WDOG

void WDOG_Deinit(WDOG_Type *base)

Shuts down the WDOG.

This function shuts down the WDOG. Ensure that the WDOG_STCTRLH.ALLOWUPDATE is 1 which indicates that the register update is enabled.

void WDOG_SetTestModeConfig(WDOG_Type *base, *wdog_test_config_t* *config)

Configures the WDOG functional test.

This function is used to configure the WDOG functional test. When called, the WDOG goes into test mode and runs according to the configuration. Ensure that the WDOG_STCTRLH.ALLOWUPDATE is 1 which means that the register update is enabled.

This is an example.

```
wdog_test_config_t test_config;  
test_config.testMode = kWDOG_QuickTest;  
test_config.timeoutValue = 0xfffffu;  
WDOG_SetTestModeConfig(wdog_base, &test_config);
```

Parameters

- base – WDOG peripheral base address
- config – The functional test configuration of WDOG

static inline void WDOG_Enable(WDOG_Type *base)

Enables the WDOG module.

This function write value into WDOG_STCTRLH register to enable the WDOG, it is a write-once register, make sure that the WCT window is still open and this register has not been written in this WCT while this function is called.

Parameters

- base – WDOG peripheral base address

static inline void WDOG_Disable(WDOG_Type *base)

Disables the WDOG module.

This function writes a value into the WDOG_STCTRLH register to disable the WDOG. It is a write-once register. Ensure that the WCT window is still open and that register has not been written to in this WCT while the function is called.

Parameters

- base – WDOG peripheral base address

static inline void WDOG_EnableInterrupts(WDOG_Type *base, uint32_t mask)

Enables the WDOG interrupt.

This function writes a value into the WDOG_STCTRLH register to enable the WDOG interrupt. It is a write-once register. Ensure that the WCT window is still open and the register has not been written to in this WCT while the function is called.

Parameters

- base – WDOG peripheral base address
- mask – The interrupts to enable The parameter can be combination of the following source if defined.
 - kWDOG_InterruptEnable

static inline void WDOG_DisableInterrupts(WDOG_Type *base, uint32_t mask)

Disables the WDOG interrupt.

This function writes a value into the WDOG_STCTRLH register to disable the WDOG interrupt. It is a write-once register. Ensure that the WCT window is still open and the register has not been written to in this WCT while the function is called.

Parameters

- base – WDOG peripheral base address
- mask – The interrupts to disable The parameter can be combination of the following source if defined.
 - kWDOG_InterruptEnable

uint32_t WDOG_GetStatusFlags(WDOG_Type *base)

Gets the WDOG all status flags.

This function gets all status flags.

This is an example for getting the Running Flag.

```
uint32_t status;
status = WDOG_GetStatusFlags(wdog_base) & kWDOG_RunningFlag;
```

See also:

_wdog_status_flags_t

- true: a related status flag has been set.
- false: a related status flag is not set.

Parameters

- base – WDOG peripheral base address

Returns

State of the status flag: asserted (true) or not-asserted (false).

void WDOG_ClearStatusFlags(WDOG_Type *base, uint32_t mask)

Clears the WDOG flag.

This function clears the WDOG status flag.

This is an example for clearing the timeout (interrupt) flag.

```
WDOG_ClearStatusFlags(wdog_base, kWDOG_TimeoutFlag);
```

Parameters

- base – WDOG peripheral base address
- mask – The status flags to clear. The parameter could be any combination of the following values. kWDOG_TimeoutFlag

```
static inline void WDOG_SetTimeoutValue(WDOG_Type *base, uint32_t timeoutCount)
```

Sets the WDOG timeout value.

This function sets the timeout value. It should be ensured that the time-out value for the WDOG is always greater than $2 \times \text{WCT time} + 20$ bus clock cycles. This function writes a value into WDOG_TOVALH and WDOG_TOVALL registers which are write-once. Ensure the WCT window is still open and the two registers have not been written to in this WCT while the function is called.

Parameters

- base – WDOG peripheral base address
- timeoutCount – WDOG timeout value; count of WDOG clock tick.

```
static inline void WDOG_SetWindowValue(WDOG_Type *base, uint32_t windowValue)
```

Sets the WDOG window value.

This function sets the WDOG window value. This function writes a value into WDOG_WINH and WDOG_WINL registers which are write-once. Ensure the WCT window is still open and the two registers have not been written to in this WCT while the function is called.

Parameters

- base – WDOG peripheral base address
- windowValue – WDOG window value.

```
static inline void WDOG_Unlock(WDOG_Type *base)
```

Unlocks the WDOG register written.

This function unlocks the WDOG register written. Before starting the unlock sequence and following configuration, disable the global interrupts. Otherwise, an interrupt may invalidate the unlocking sequence and the WCT may expire. After the configuration finishes, re-enable the global interrupts.

Parameters

- base – WDOG peripheral base address

```
void WDOG_Refresh(WDOG_Type *base)
```

Refreshes the WDOG timer.

This function feeds the WDOG. This function should be called before the WDOG timer is in timeout. Otherwise, a reset is asserted.

Parameters

- base – WDOG peripheral base address

```
static inline uint16_t WDOG_GetResetCount(WDOG_Type *base)
```

Gets the WDOG reset count.

This function gets the WDOG reset count value.

Parameters

- base – WDOG peripheral base address

Returns

WDOG reset count value.

```
static inline void WDOG_ClearResetCount(WDOG_Type *base)
```

Clears the WDOG reset count.

This function clears the WDOG reset count value.

Parameters

- base – WDOG peripheral base address

FSL_WDOG_DRIVER_VERSION

Defines WDOG driver version 2.0.2.

WDOG_FIRST_WORD_OF_UNLOCK

First word of unlock sequence

WDOG_SECOND_WORD_OF_UNLOCK

Second word of unlock sequence

WDOG_FIRST_WORD_OF_REFRESH

First word of refresh sequence

WDOG_SECOND_WORD_OF_REFRESH

Second word of refresh sequence

enum __wdog_clock_source

Describes WDOG clock source.

Values:

enumerator kWDog_LpoClockSource

WDog clock sourced from LPO

enumerator kWDog_AlternateClockSource

WDog clock sourced from alternate clock source

enum __wdog_clock_prescaler

Describes the selection of the clock prescaler.

Values:

enumerator kWDog_ClockPrescalerDivide1

Divided by 1

enumerator kWDog_ClockPrescalerDivide2

Divided by 2

enumerator kWDog_ClockPrescalerDivide3

Divided by 3

enumerator kWDog_ClockPrescalerDivide4

Divided by 4

enumerator kWDog_ClockPrescalerDivide5

Divided by 5

enumerator kWDog_ClockPrescalerDivide6

Divided by 6

enumerator kWDog_ClockPrescalerDivide7

Divided by 7

enumerator kWDog_ClockPrescalerDivide8

Divided by 8

enum __wdog_test_mode

Describes WDOG test mode.

Values:

enumerator kWDog_QuickTest

Selects quick test

enumerator kWDOG_ByteTest

Selects byte test

enum __wdog_tested_byte

Describes WDOG tested byte selection in byte test mode.

Values:

enumerator kWDOG_TestByte0

Byte 0 selected in byte test mode

enumerator kWDOG_TestByte1

Byte 1 selected in byte test mode

enumerator kWDOG_TestByte2

Byte 2 selected in byte test mode

enumerator kWDOG_TestByte3

Byte 3 selected in byte test mode

enum __wdog_interrupt_enable_t

WDOG interrupt configuration structure, default settings all disabled.

This structure contains the settings for all of the WDOG interrupt configurations.

Values:

enumerator kWDOG_InterruptEnable

WDOG timeout generates an interrupt before reset

enum __wdog_status_flags_t

WDOG status flags.

This structure contains the WDOG status flags for use in the WDOG functions.

Values:

enumerator kWDOG_RunningFlag

Running flag, set when WDOG is enabled

enumerator kWDOG_TimeoutFlag

Interrupt flag, set when an exception occurs

typedef enum __wdog_clock_source wdog_clock_source_t

Describes WDOG clock source.

typedef struct __wdog_work_mode wdog_work_mode_t

Defines WDOG work mode.

typedef enum __wdog_clock_prescaler wdog_clock_prescaler_t

Describes the selection of the clock prescaler.

typedef struct __wdog_config wdog_config_t

Describes WDOG configuration structure.

typedef enum __wdog_test_mode wdog_test_mode_t

Describes WDOG test mode.

typedef enum __wdog_tested_byte wdog_tested_byte_t

Describes WDOG tested byte selection in byte test mode.

typedef struct __wdog_test_config wdog_test_config_t

Describes WDOG test mode configuration structure.

WDOG_WCT_INSTRUCITON_COUNT

< Watchdog configuration time window

struct __wdog_work_mode

#include <fsl_wdog.h> Defines WDOG work mode.

Public Members

bool enableWait

Enables or disables WDOG in wait mode

bool enableStop

Enables or disables WDOG in stop mode

bool enableDebug

Enables or disables WDOG in debug mode

struct __wdog_config

#include <fsl_wdog.h> Describes WDOG configuration structure.

Public Members

bool enableWdog

Enables or disables WDOG

wdog_clock_source_t clockSource

Clock source select

wdog_clock_prescaler_t prescaler

Clock prescaler value

wdog_work_mode_t workMode

Configures WDOG work mode in debug stop and wait mode

bool enableUpdate

Update write-once register enable

bool enableInterrupt

Enables or disables WDOG interrupt

bool enableWindowMode

Enables or disables WDOG window mode

uint32_t windowValue

Window value

uint32_t timeoutValue

Timeout value

struct __wdog_test_config

#include <fsl_wdog.h> Describes WDOG test mode configuration structure.

Public Members

wdog_test_mode_t testMode

Selects test mode

wdog_tested_byte_t testedByte

Selects tested byte in byte test mode

uint32_t timeoutValue
Timeout value

2.54 XBAR: Inter-Peripheral Crossbar Switch

void XBAR_Init(XBAR_Type *base)

Initializes the XBAR modules.

This function un-gates the XBAR clock.

Parameters

- base – XBAR peripheral address.

void XBAR_Deinit(XBAR_Type *base)

Shutdown the XBAR modules.

This function disables XBAR clock.

Parameters

- base – XBAR peripheral address.

void XBAR_SetSignalsConnection(XBAR_Type *base, xbar_input_signal_t input,
xbar_output_signal_t output)

Set connection between the selected XBAR_IN[*] input and the XBAR_OUT[*] output signal.

This function connects the XBAR input to the selected XBAR output. If more than one XBAR module is available, only the inputs and outputs from the same module can be connected.

Example:

```
XBAR_SetSignalsConnection(XBAR, kXBAR_InputTMR_CH0_Output, kXBAR_OutputXB_DMA_
↪INT2);
```

Parameters

- base – XBAR peripheral address
- input – XBAR input signal.
- output – XBAR output signal.

void XBAR_ClearStatusFlags(XBAR_Type *base, uint32_t mask)

Clears the edge detection status flags of relative mask.

Parameters

- base – XBAR peripheral address
- mask – the status flags to clear.

uint32_t XBAR_GetStatusFlags(XBAR_Type *base)

Gets the active edge detection status.

This function gets the active edge detect status of all XBAR_OUTs. If the active edge occurs, the return value is asserted. When the interrupt or the DMA functionality is enabled for the XBAR_OUTx, this field is 1 when the interrupt or DMA request is asserted and 0 when the interrupt or DMA request has been cleared.

Example:

```
uint32_t status;

status = XBAR_GetStatusFlags(XBAR);
```

Parameters

- base – XBAR peripheral address.

Returns

the mask of these status flag bits.

```
void XBAR_SetOutputSignalConfig(XBAR_Type *base, xbar_output_signal_t output, const
                               xbar_control_config_t *controlConfig)
```

Configures the XBAR control register.

This function configures an XBAR control register. The active edge detection and the DMA/IRQ function on the corresponding XBAR output can be set.

Example:

```
xbar_control_config_t userConfig;
userConfig.activeEdge = kXBAR_EdgeRising;
userConfig.requestType = kXBAR_RequestInterruptEnalbe;
XBAR_SetOutputSignalConfig(XBAR, kXBAR_OutputXB_DMA_INT0, &userConfig);
```

Parameters

- base – XBAR peripheral address
- output – XBAR output number.
- controlConfig – Pointer to structure that keeps configuration of control register.

```
enum _xbar_active_edge
```

XBAR active edge for detection.

Values:

```
enumerator kXBAR_EdgeNone
```

Edge detection status bit never asserts.

```
enumerator kXBAR_EdgeRising
```

Edge detection status bit asserts on rising edges.

```
enumerator kXBAR_EdgeFalling
```

Edge detection status bit asserts on falling edges.

```
enumerator kXBAR_EdgeRisingAndFalling
```

Edge detection status bit asserts on rising and falling edges.

```
enum _xbar_request
```

Defines the XBAR DMA and interrupt configurations.

Values:

```
enumerator kXBAR_RequestDisable
```

Interrupt and DMA are disabled.

```
enumerator kXBAR_RequestDMAEnable
```

DMA enabled, interrupt disabled.

```
enumerator kXBAR_RequestInterruptEnalbe
```

Interrupt enabled, DMA disabled.

```
enum _xbar_status_flag_t
```

XBAR status flags.

This provides constants for the XBAR status flags for use in the XBAR functions.

Values:

enumerator kXBAR_EdgeDetectionOut0

XBAR_OUT0 active edge interrupt flag, sets when active edge detected.

typedef enum *_xbar_active_edge* xbar_active_edge_t

XBAR active edge for detection.

typedef enum *_xbar_request* xbar_request_t

Defines the XBAR DMA and interrupt configurations.

typedef enum *_xbar_status_flag_t* xbar_status_flag_t

XBAR status flags.

This provides constants for the XBAR status flags for use in the XBAR functions.

typedef struct *_xbar_control_config* xbar_control_config_t

Defines the configuration structure of the XBAR control register.

This structure keeps the configuration of XBAR control register for one output. Control registers are available only for a few outputs. Not every XBAR module has control registers.

FSL_XBAR_DRIVER_VERSION

XBAR_SELx(base, output)

XBAR_WR_SELx_SELx(base, input, output)

struct *_xbar_control_config*

#include <fsl_xbar.h> Defines the configuration structure of the XBAR control register.

This structure keeps the configuration of XBAR control register for one output. Control registers are available only for a few outputs. Not every XBAR module has control registers.

Public Members

xbar_active_edge_t activeEdge

Active edge to be detected.

xbar_request_t requestType

Selects DMA/Interrupt request.

Chapter 3

Middleware

3.1 Motor Control

3.1.1 FreeMASTER

Communication Driver User Guide

Introduction

What is FreeMASTER? FreeMASTER is a PC-based application developed by NXP for NXP customers. It is a versatile tool usable as a real-time monitor, visualization tool, and a graphical control panel of embedded applications based on the NXP processing units.

This document describes the embedded-side software driver which implements an interface between the application and the host PC. The interface covers the following communication:

- **Serial** UART communication either over plain RS232 interface or more typically over a USB-to-Serial either external or built in a debugger probe.
- **USB** direct connection to target microcontroller
- **CAN bus**
- **TCP/IP network** wired or WiFi
- **Segger J-Link RTT**
- **JTAG** debug port communication
- ...and all of the above also using a **Zephyr** generic drivers.

The driver also supports so-called “packet-driven BDM” interface which enables a protocol-based communication over a debugging port. The BDM stands for Background Debugging Module and its physical implementation is different on each platform. Some platforms leverage a semi-standard JTAG interface, other platforms provide a custom implementation called BDM. Regardless of the name, this debugging interface enables non-intrusive access to the memory space while the target CPU is running. For basic memory read and write operations, there is no communication driver required on the target when communicating with the host PC. Use this driver to get more advanced FreeMASTER protocol features over the BDM interface. The driver must be configured for the packet-driven BDM mode, in which the host PC uses the debugging interface to write serial command frames directly to the target memory buffer. The same method is then used to read response frames from that memory buffer.

Similar to “packet-driven BDM”, the FreeMASTER also supports a communication over [J-Link RTT](<https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/>) interface defined by SEGGER Microcontroller GmbH for ARM CortexM-based microcontrollers. This method also uses JTAG physical interface and enables high-speed real time communication to run over the same channel as used for application debugging.

Driver version 3 This document describes version 3 of the FreeMASTER Communication Driver. This version features the implementation of the new Serial Protocol, which significantly extends the features and security of its predecessor. The new protocol internal number is v4 and its specification is available in the documentation accompanying the driver code.

Driver V3 is deployed to modern 32-bit MCU platforms first, so the portfolio of supported platforms is smaller than for the previous V2 versions. It is recommended to keep using the V2 driver for legacy platforms, such as S08, S12, ColdFire, or Power Architecture. Reach out to [FreeMASTER community](#) or to the local NXP representative with requests for more information or to port the V3 driver to legacy MCU devices.

Thanks to a layered approach, the new driver simplifies the porting of the driver to new UART, CAN or networking communication interfaces significantly. Users are encouraged to port the driver to more NXP MCU platforms and contribute the code back to NXP for integration into future releases. Existing code and low-level driver layers may be used as an example when porting to new targets.

Note: Using the FreeMASTER tool and FreeMASTER Communication Driver is only allowed in systems based on NXP microcontroller or microprocessor unit. Use with non-NXP MCU platforms is **not permitted** by the license terms.

Target platforms The driver implementation uses the following abstraction mechanisms which simplify driver porting and supporting new communication modules:

- **General CPU Platform** (see source code in the `src/platforms` directory). The code in this layer is only specific to native data type sizes and CPU architectures (for example; alignment-aware memory copy routines). This driver version brings two generic implementations of 32-bit platforms supporting both little-endian and big-endian architectures. There are also implementations customized for the 56F800E family of digital signal controllers and S12Z MCUs. **Zephyr** is treated as a specific CPU platform as it brings unified user configuration (Kconfig) and generic hardware device drivers. With Zephyr, the transport layer and low-level communication layers described below are configured automatically using Kconfig and Device Tree technologies.
- **Transport Communication Layer** - The Serial, CAN, Networking, PD-BDM, and other methods of transport logic are implemented as a driver layer called `FMSTR_TRANSPORT` with a uniform API. A support of the Network transport also extends single-client modes of operation which are native for Serial, USB and CAN by a concept of multiple client sessions.
- **Low-level Communication Driver** - Each type of transport further defines a low-level API used to access the physical communication module. For example, the Serial transport defines a character-oriented API implemented by different serial communication modules like UART, LPUART, USART, and also USB-CDC. Similarly, the CAN transport defines a message-oriented API implemented by the FlexCAN or MCAN modules. Moreover, there are multiple different implementations for the same kind of communication peripherals. The difference between the implementation is in the way the low-level hardware registers are accessed. The `mcuxsdk` folder contains implementations which use MCUXpresso SDK drivers. These drivers should be used in applications based on the NXP MCUXpresso SDK. The “ampsdk” drivers target automotive-specific MCUs and their respective SDKs. The “dreg” implementations use a plain C-language access to hardware register addresses which makes it a universal and the most portable solution. In this case, users are encouraged to add more drivers for other communication modules or other respective SDKs and contribute the code back to NXP for integration.

The low-level drivers defined for the Networking transport enable datagram-oriented UDP and stream TCP communication. This implementation is demonstrated using the lwIP software stack but shall be portable to other TCP/IP stacks. It may sound surprisingly, but also the Segger J-Link RTT communication driver is linked to the Networking transport (RTT is stream oriented communication handled similarly to TCP).

Replacing existing drivers For all supported platforms, the driver described in this document replaces the V2 implementation and also older driver implementations that were available separately for individual platforms (PC Master SCI drivers).

Clocks, pins, and peripheral initialization The FreeMASTER communication driver is only responsible for runtime processing of the communication and must be integrated with an user application code to function properly. The user application code is responsible for general initialization of clock sources, pin multiplexers, and peripheral registers related to the communication speed. Such initialization should be done before calling the FMSTR_Init function.

It is recommended to develop the user application using one of the Software Development Kits (SDKs) available from third parties or directly from NXP, such as MCUXpresso SDK, MCUXpresso IDE, and related tools. This approach simplifies the general configuration process significantly.

MCUXpresso SDK The MCUXpresso SDK is a software package provided by NXP which contains the device initialization code, linker files, and software drivers with example applications for the NXP family of MCUs. The MCUXpresso Config Tools may be used to generate the clock-setup and pin-multiplexer setup code suitable for the selected processor.

The MCUXpresso SDK also contains this FreeMASTER communication driver as a “middleware” component which may be downloaded along with the example applications from <https://mcuxpresso.nxp.com/en/welcome>.

MCUXpresso SDK on GitHub The FreeMASTER communication driver is also released as one of the middleware components of the MCUXpresso SDK on the GitHub. This release enables direct integration of the FreeMASTER source code Git repository into a target applications including Zephyr applications.

Related links:

- [The official FreeMASTER middleware repository.](#)
- [Online version of this document](#)

FreeMASTER in Zephyr The FreeMASTER middleware repository can be used with MCUXpresso SDK as well as a Zephyr module. Zephyr-specific samples which include examples of Kconfig and Device Tree configurations for Serial, USB and Network communications are available in separate repository. West manifest in this sample repository fetches the full Zephyr package including the FreeMASTER middleware repository used as a Zephyr module.

Example applications

MCUX SDK Example applications There are several example applications available for each supported MCU platform.

- **fmstr_uart** demonstrates a plain serial transmission, typically connecting to a computer’s physical or virtual COM port. The typical transmission speed is 115200 bps.

- **fmstr_can** demonstrates CAN bus communication. This requires a suitable CAN interface connected to the computer and interconnected with the target MCU using a properly terminated CAN bus. The typical transmission speed is 500 kbps. A FreeMASTER-over-CAN communication plug-in must be used.
- **fmstr_usb_cdc** uses an on-chip USB controller to implement a CDC communication class. It is connected directly to a computer's USB port and creates a virtual COM port device. The typical transmission speed is above 1 Mbps.
- **fmstr_net** demonstrates the Network communication over UDP or TCP protocol. Existing examples use lwIP stack to implement the communication, but in general, it shall be possible to use any other TCP/IP stack to achieve the same functionality.
- **fmstr_wifi** is the fmstr_net application modified to use a WiFi network interface instead of a wired Ethernet connection.
- **fmstr_rtt** demonstrates the communication over SEGGER J-Link RTT interface. Both fmstr_net and fmstr_rtt examples require the FreeMASTER TCP/UDP communication plug-in to be used on the PC host side.
- **fmstr_eonce** uses the real-time data unit on the JTAG EOnCE module of the 56F800E family to implement pseudo-serial communication over the JTAG port. The typical transmission speed is around 10 kbps. This communication requires FreeMASTER JTAG/EOnCE communication plug-in.
- **fmstr_pdbdm** uses JTAG or BDM debugging interface to access the target RAM directly while the CPU is running. Note that such approach can be used with any MCU application, even without any special driver code. The computer reads from and writes into the RAM directly without CPU intervention. The Packet-Driven BDM (PD-BDM) communication uses the same memory access to exchange command and response frames. With PD-BDM, the FreeMASTER tool is able to go beyond basic memory read/write operations and accesses also advanced features like Recorder, TSA, or Pipes. The typical transmission speed is around 10 kbps. A PD-BDM communication plug-in must be used in FreeMASTER and configured properly for the selected debugging interface. Note that this communication cannot be used while a debugging interface is used by a debugger session.
- **fmstr_any** is a special example application which demonstrates how the NXP MCUXpresso Config Tools can be used to configure pins, clocks, peripherals, interrupts, and even the FreeMASTER “middleware” driver features in a graphical and user friendly way. The user can switch between the Serial, CAN, and other ways of communication and generate the required initialization code automatically.

Zephyr sample 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.

Chapter 4

RTOS

4.1 FreeRTOS

4.1.1 FreeRTOS kernel

Open source RTOS kernel for small devices.

[FreeRTOS kernel for MCUXpresso SDK Readme](#)

[FreeRTOS kernel for MCUXpresso SDK ChangeLog](#)

[FreeRTOS kernel Readme](#)

4.1.2 FreeRTOS drivers

This is set of NXP provided FreeRTOS reentrant bus drivers.

4.1.3 backoffalgorithm

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

[Readme](#)

4.1.4 corehttp

C language HTTP client library designed for embedded platforms.

4.1.5 corejson

JSON parser.

Readme

4.1.6 coremqtt

MQTT publish/subscribe messaging library.

4.1.7 corepkcs11

PKCS #11 key management library.

Readme

4.1.8 freertos-plus-tcp

Open source RTOS FreeRTOS Plus TCP.

Readme